

PostgreSQL Bootcamp: Beginner to Advanced

Learn about Subqueries, CTEs,
Indexes, PL/SQL, PL/pgSQL,
Triggers, Cursors, Partitions,
Views, JSON, Window Functions,
DBA tasks

1. Sample Movies Database & setup:

the movies database overview | the actors, directors, movies & revenues tables | create movies database tables | import movies database table data |

2. Modifying Data in the Tables:

inserting single & multiple rows | escaping apostrophe in a string | returning clause | delete row | delete all rows | upsert on conflict do nothing | upsert on conflict update rows

3. Querying Data from a Database:

aliasing with & without AS keyword | aliasing as a string | concatenating columns in a query | order by asc & desc | order by nulls first & last | select distinct single or multiple columns or all

4. Filtering Data:

Comparison, Logical & Arithmetic Operators overview | And Or Operator, Combining And with OR Operator, order of execution | Where Clause Order of Operation & Alias in Where Clause | Using Comparison Operators with numeric, dates, text | Using Limit & Offset | Using Fetch | Using IN & Not IN | Using BETWEEN & Not BETWEEN | Using LIKE & ILIKE | Using Is NULL & IS NOT NULL | Concatenation, CONCAT & CONCAT_WS | Null values with ||, CONCAT & CONCAT_WS

5. PostgreSQL Data Types:

Boolean & accepted values | CHAR, VARCHAR, TEXT differences & use cases | integers - SMALLINT, INT, BIGINT & use cases | decimal/numeric, Real & Double precision & use cases | Default behaviours of Numeric, Real & Double Precision | Date and Time - Data, Time, Timestamp & TimestampTZ & use cases | interval function w/ date/time | timestampTZ nuances |

UUID Data Type | hstore & selecting from a key=>value | JSON | JSONB & GIN indexing | network address data type

6. Modify Tables, Constraints & Type Conversion:

Add/modify Table Columns | cast using:: | Add Check or unique constraint to a column | implicit and explicit data conversion | explicit conversion of string to int, string to bool, string to double precision | string to date, timestamp & timestamptz | string to interval | implicit to explicit data conversions | data conversion using case statement | TO_CHAR function | TO_NUMBER function | TO_DATE function | TO_TIMESTAMP function

7. User Defined Data Types:

Domain Data types: address example, positive number example, email validation regex example, enum set of values, list all domain data types, Drop domain type | Create composite address object | composite inventory item example | Enum data type and dropping | alter, rename, set schema and add attribute to data type | Alter enum type in production server | Enum with default value in table | Create a Type If Not Exists Using PL/pgSQL Function

8. PostgreSQL Constraints:

Constraints: not null | unique | default | primary key | composite primary key | foreign key | check | drop a constraint | drop a FK constraint

9. PostgreSQL Sequences:

create sequence | advance a Sequence | get current value | set value | restart & rename sequence | use PgAdmin 4 to alter a sequence | start with | increment | MinValue &

alter a sequence | start with | increment | MinValue & MaxValue | sequence data type cycle sequence | delete sequence | sequence datatype | Descending sequence | cycle sequence | delete sequence | attach sequence to table column | list all sequences in a database | share one sequence between two tables | create an alphanumeric sequence

10. String Functions:

upper | lower | intcap | left | right | reverse | split_part | trim | btrim | ltrim | rtrim | lpad | rpad | length | char_length | position | strpos | substring | repeat | replace

11. Aggregate Functions:

count() | count(*) | count(1) | count with distinct | sum | sum with distinct | min | max | greatest | least | greatest() vs max() | avg | avg with null values | combining columns with mathematical operators

12. Using Date/Time Functions:

to_date() | to_timestamp() | to_char() | to_date() | to_timestamp() | make_date() | make_time() | make_timestamp() | make_interval() | make_timestamptz() | extract() | date_part() | math operators with date/time/timestamp/interval | ()overlaps() | current_date(), current_time(), current_timestamp(), localtime(), (), localtimestamp(), now(), transaction_timestamp() | statement_timestamp() | clock_timestamp() | timeofday() | age() | date accuracy with epoch | pg_timezone_names | pg_timezone_abbrevs | show time zone | set time zone | handling date and time fields in a database | date_part()

13. Grouping Data:

group by | group by multiple columns | having | sql clauses

order of execution | having vs where | null values with group by using coalesce

14. Joining Multiple Tables:

inner join | aliasing tbl names | on clause | join clause | alias.* | inner join using | joining 2+ tables | using cast to join on dissimilar data types | left joins | right joins | full joins | self join | cross join | natural join | using coalesce to combine data from two tables | on vs where

15. Combining Queries Together:

union | union all | union matched data types | union with filters & conditions | union with different number of columns | intersect | except

16. PostgreSQL Schemas:

what is a schema | add,delete,alter schema | schema hierarchy | move table to new schema | schema search path | duplicate schema and data pg_dump | system catalogue pg_catalog | compare tables and columns in two schemas | schemas and privileges

17. Exploring Array Functions:

int4range, numrange, tsrange, tstzrange, daterange open - closed | arrays | comparison operators with arrays & ranges | Inclusion operators with arrays & ranges | array constructions with concat ||, array_cat(), array_prepend(), array_append() | array_ndims() | array_dims() | array_length() | array_lower() | array_upper() | cardinality() | array search functions: array_position() & array_positions() | array_remove() | array_to_string() | string_to_array | using arrays in table columns | insert & query array data | update array data | array dimensions are ignored! | unest() | multi-dimensional arrays | array vs jsonb |

columns | insert & query array data | update array data | array dimensions are ignored! | unest() | multi-dimensional arrays | array vs jsonb |

18. JSON with PostgreSQL:

what is JSON? | JSON syntax | JSON & JSONB data types | representing JSON in postgresql | Creating tables with JSONB columns | select named pairs from JSONB data | update named:pairs: delete pair, update values | insert nested arrays into JSONB | remove elements from nested array in JSONB | row_to_json() | json_agg() | json_build_array() | json_build_object() | json_objects() | handling JSON nulls | jsonb_array_length() | jsonb_each() | jsonb_to_record() | existence operator | containment operator | Search JSON with postgresql functions | indexing JSONB with GIN

19. Indexes & Performance Optimisation:

what is an index? | create an index (single-column, multi-column) sql|pgAdmin4 | create unique indexes (single-column, multi-column) | list all indexes | index size | pg_indexes_size() | pg_stat_all_indexes() | drop an index | SQL statement execution process | SQL statement execution stages | query optimizer | optimizer node types | Query cost analysis: seq nodes | index nodes | index only nodes | Bitmap Index nodes | hash joins | merge joins | nested loop joins | working memory | b-tree index | hash index | brin index | gin index | explain statement | brin index | gin index | explain statement | explain output options | explain analyze | query cost model & formula | cost of indexes (table size, index size, query speed) | indexes for sorted output | execution plans depend on input values | Using Organized Vs Random Data | index only vs index scan | query plan speed | partial indexes | expression index | adding data while indexing | invalidating an index | re-index

19. Indexes & Performance Optimisation:

what is an index? | create an index (single-column, multi-column) sql|pgAdmin4 | create unique indexes (single-column multi-column) | list all indexes | index size | pg_indexes_size() | pg_stat_all_indexes() | drop an index | SQL statement execution process | SQL statement execution stages | query optimizer | optimizer node types | Query cost analysis: seq nodes | index nodes | index only nodes | Bitmap Index nodes | hash joins | merge joins | nested loop joins | working memory | b-tree index | hash index | brin index | gin index | explain statement | brin index | gin index | explain statement | explain output options | explain analyze | query cost model & formula | cost of indexes (table size, index size, query speed) | indexes for sorted output | execution plans depend on input values | Using Organized Vs Random Data | index only vs index scan | query plan speed | partial indexes | expression index | adding data while indexing | invalidating an index | re-index

20. All About Views:

what are views? | create view | rename view | delete view | create view with filters | using union joins in a view | using inner joins in a view | column order in a view | append a column to view | delete column from view? | regular view does not store data | updatable views | CRUD in updatable views | updatable view check option check local | check cascade | what is a materialized view? | create matview | drop materialized view | changing materialized view data | populate matview in pg_class? | concurrent refresh mview | materialized views pros and cons | example of using matview for website stat analysis | pg query to find all matviews with no unique index | other useful pg matview queries array_replace() | in & not in | all | any | some

21. SubQueries:

Subqueries | where | IN | subqueries with JOIN | using union without order by | subqueries with alias | select without from clause | correlated queries | select.. in(subquery) | any with subquery | all with subquery | subquery using exists

22. Common Table Expressions (CTE):

what is a cte | how to build a cte | using case statement in cte | join cte result set with table | simultaneous delete & insert with cte | recursive cte for parent child relationship

23. Grouping Sets:

summarisation with grouping | group by with rollup | using grouping, coalesce & case to produce a beatified rollup table

24. Window Functions:

Intro to windows functions | prepare & understand global trades sample data | use: Agg functions, rollup on single & multiple columns, group by cube & grouping sets | group by cube | aggregation query performance | filter clause | Using Window Function | over clause | sorting inside a window | ordering inside a window | sliding dynamic window | understanding window framing: rows, Between proceeding, unbound, current | rows & ranges | using multiple windows in one query | ntile buckets | lead & lag function | last_value() | first_value() | nth_value() | corr() | row_number() window with partitions | row_number() with order by | row_number() within subquery to find nth record of each partition | apply row_number() to distinct rows | over function calculate percentage | compare difference to avg | cumulative totals | lead & lag compare next and prev values | compare with largest & smallest | window functions sumary |

planning tips for window functions | rank vs dense rank | compute global and group ranks | partition by group averages | order by our own defined values | delete & insert in one query using with

25. Using Regular Expressions for Text Patterns:

intro to regex | regular expressions notations table | similar to | POSIX ~, ~*, !~, !~* | Using regular expression with substring() | regexp_matches() | regexp_replace() | regexp_split_to_table() | regexp_split_to_array()

26. Powerful Text Searches in PostGreSQL:

drawbacks of %LIKE% based search queries | what is full text search | to_tsvector() | to_tsquery | build tsvector table | apply GIN index on tsvector | ts_headline | ts_rank()

27. Table Partitions:

what is a partition? | when a partition should be used | table inheritance theory | partition types: range, list & hash | partition by range CRUD | partition by list CRUD | partition by hash CRUD | default partition | multi-level partitioning | add new partition | detach partition | alter partition bounds within a transaction | partition_pruning | choseing a field to partition over | dimensioning the partition | partition advantages | common partition mistakes

28. Server PProgramming:

postgreSQL as a development platform? | procedural languages | keep the data on the server! | functions vs stored procedures | user-defined functions | structure of a function

29. Functions with SQL language:

sql function syntax | create sql function | view function details in pgadmin4 | \$\$ and \$body\$ quotation | functions that return void | functions that return a single value | functions using parameters | a function that returns a composite (array of row data) .* or .column_name | select columns from a function return | return a table from a function | function default parameters | function based on views | drop a function

30. Functions with PL/pgSQL language:

PL/pgSQL introduction | PL/pgSQL vs SQL | structure of a PL/pgSQL function | Declaring variables, using alias for | declaring variables in a function | variables initializing timing | using IN, OUT without RETURNS | scope of variables in block and subblock | how to return query results | Control Structures: IF statements, using if with table data, CASE statements (simple & searched) | loop statement | for loop | iterate over result set in for loop | continue statement | foreach loop | while loop | using return query | returning a table | return next | error and exception handling | **eh – too_mny_rows** | **sqlstate codes for eh** | **data exception errors**

31. Exploring Stored Procedures:

functions vs stored procedures | create transactions | understand use of stored procedures | Returning a value from stored procedure | drop a stored procedure

32. PostGreSQL Triggers:

intro to triggers | what is a trigger? | types of trigger | trigger table | pros & cons of triggers | trigger key points | steps to create a trigger | create a data auditing trigger | trigger to modify data at insert event |

a triggered function to view trigger variables | trigger function to disallow DELETE on a table | trigger function to disallow TRUNCATE on a table | creating an audit trigger | creating conditional triggers | disallow data change on primary key | use triggers very cautiously | what is an event trigger | event triggers usage scenarios | creating event triggers | event trigger events & variables | creating and audit event trigger | prevent schema changes | dropping a trigger

33. Using Cursors:

understanding row by row operations | cursors and procedural language | steps to create a cursor | creating a cursor | opening a cursor | using a cursor | updating data | closing a cursor | creating a PL/pgSQL cursor | using a parametric cursor via function

34. Creating Crosstab Reports:

what is a crosstab report? | installing the tablefunc extension | creating a crosstab report | order matters in crosstab reports | pivoting with sample rainfalls data | pivoting rows and columns | matrix report via a query | aggregate over a filter | static to dynamic pivots | dynamic pivot query via json | dynamic pivot table columns | interactive client-side pivot | handling missing values

35. Internationalisation:

global character set support | client and server encoding

36. PostGresSQL Transactions:

what is a transaction? | how SQL protects database during transaction | ACID database | transaction setup | transaction analysis | how to fix aborted transactions

how to fix transactions on a crash | partial transaction rollback with savepoints | using savepoints with transactions | how to fix aborted transaction | how to fix transaction on crash | partial transaction rollback with savepoints | using SAVEPOINT with transaction

37. Using Northwinds Database:

orders shipping to USA or france | total number of orders shipped to USA or france | orders shipped to latin america | show total order amount for each order | first the oldest and latest order date | total products in each category | list products that need reordering | freight analysis | customers with no orders | top customers with total orders amount | orders with many lines of ordered items | orders with double entry items | late shipped orders by employees | countries with customers or suppliers (union & cte) | customers with multiple orders in 4 day period | first order from each country

38. Using HR Database:

human resources database structure | quick overview of all tables | get highest, lowest, total & average salaries of employees | difference between highest & lowest salaries by job_id | get lowest paid salaries by each manager | average salary for each department with more than 10 employees | average salary for each department excluding programmers | max salary for each post where salary is equal or above \$5000 | using alias name with columns | compute 15% of salaries for all employees | list all employee ids with each job id group | discard characters from employee email address | list all employees with first name beginning with A, B or M | string manipulation with upper, lower & initcap functions | using substring to get portion of string data | get unique designations in employees table |

get employees with particular department id | select employees not in range | find 'c' in last name at 3rd or greater position | update phone number with strings | get monthly salary of each employee | calculate average salary with total number of employees | find employees with 6 letter names

39. Exploring Stock Markets Data:

select first or last records in a table | get the first or last record for each group | how to calculate cube root in PostgreSQL

40. Managing PostgreSQL security:

security concepts & access levels | instance level security | add users to roles | use pgAdmin4 to create roles | database level security | schema level security | table level security | column level security | row level security | using CURRENT_USER with rls | row level policy for application users (session) | drop a RLS policy | inspecting table permissions | row level security performance | encrypted data in columns

41. PostgreSQL Utilities - psql:

connect to a local or remote database | switch connection to a new database | list all databases & tables | list all table space, schemas & indexes | list all sequences, roles, data types & domain data types | describe a table | edit SQL commands in an editor | display command history, run commands from a file | built in syntax reference for commands | represent NULL values on psql terminal | pretty a psql table layout | repeatedly execute a command | turn on query execution time

42. PostgreSQL Internals:

list database users, & database sizes with & without indexes | list all databases & schemas | list all tables and views

list all columns from a table | view system metadata via system
information functions | view privileges information across
tables | using system administration functions | show all
running queries | terminate running and IDLE process | how to
check live and dead rows in table | file layout of PostgreSQL
Tables

[43. Managing Tables:](#)

Using SELECT INTO to create a new table with joins | Duplicate
a Table with or without Data | Import Data from CSV file |
Export Data to CSV file | Deleting Duplicate records | Backup
Your Database Regularly | Test Queries on a Test Database |
Not Documenting your Process? | Database Operations and
Table Size | Tracking Table Size | PostgreSQL autovacuum
Process | Recovering Unused space with Vacuum | Generate
Columns

1. Sample Movies Database & setup

[1.1 The Movies Database Overview](#)

[1.2 The Actors, Directors & Movies Tables](#)

[1.3 The Revenues Table](#)

[1.4 Create Movies Database Tables](#)

[1.5 Import Movies Database Table Data](#)

[1.6 Creating databases & tables in PgAdmin 4](#)

[1.7 Altering a Table in PgAdmin 4](#)

[1.8 Using PgAdmin Query Tool](#)

1.1 The Movies Database Overview

Below are the tables for the sample movies database

▼ actors

▼ Columns (7)

- actor_id
- first_name
- last_name
- gender
- date_of_birth
- add_date
- update_date

▼ directors

▼ Columns (7)

- director_id
- first_name
- last_name
- date_of_birth
- nationality
- add_date
- update_date

▼ movies

▼ Columns (7)

- movie_id
- movie_name
- movie_length
- movie_lang
- age_certificate
- release_date
- director_id

▼ movies_actors

▼ Columns (2)

- movie_id
- actor_id

▼ movies_revenues

▼ Columns (4)

- revenue_id
- movie_id
- revenues_domestic
- revenues_international

1.2 The Actors, Directors & Movies Tables

actors

	actor_id [PK] integer	first_name character varying (150)	last_name character varying (150)	gender character (1)	date_of_birth date	add_date date	update_date date
1	1	Malin	Akerman	F	1978-05-12	[null]	[null]
2	2	Tim	Allen	M	1953-06-13	[null]	[null]
3	3	Julie	Andrews	F	1935-10-01	[null]	[null]
4	4	Ivana	Baquero	F	1994-06-11	[null]	[null]

directors

	director_id [PK] integer	first_name character varying (150)	last_name character varying (150)	date_of_birth date	nationality character varying (20)
1	1	Tomas	Alfredson	1965-04-01	Swedish
2	2	Paul	Anderson	1970-06-26	American
3	3	Wes	Anderson	1969-05-01	American
4	4	Richard	Ayoade	1977-06-12	British

movies

[Data Output](#) [Explain](#) [Messages](#) [Notifications](#)

	movie_id [PK] integer	movie_name character varying (100)	movie_length integer	movie_lang character varying (20)	age_certificate character varying (10)	release_date date	director_id integer
1	1	A Clockwork Orange	112	English	18	1972-02-02	13
2	2	Apocalypse Now	168	English	15	1979-08-15	9
3	3	Battle Royale	111	Japanese	18	2001-01-04	10
4	4	Blade Runner	121	English	15	1982-06-25	27
5	5	Chungking Express	113	Chinese	15	1996-08-03	35

1.3 The Revenues Table

Movies_revenues

	revenue_id [PK] integer	movie_id integer	revenues_domestic numeric (10,2)	revenues_international numeric (10,2)
1	1	45	22.20	1.30
2	2	13	199.40	201.20
3	3	23	102.10	[null]
4	4	44	158.70	[null]
5	6	1	27.10	[null]
6	7	53	[null]	[null]
7	17	18	260.30	210.90

1.4 Create Movies Database Tables

```
-- Create actor table
CREATE TABLE actors (
    actor_id SERIAL PRIMARY KEY,
    first_name VARCHAR(150),
    last_name VARCHAR(150) NOT NULL,
    gender CHAR (1),
    date_of_birth DATE,
    update_date DATE
);

-- Create Directors table
CREATE TABLE directors (
    director_id SERIAL PRIMARY KEY,
    first_name VARCHAR(150),
    last_name VARCHAR(150),
    date_of_birth DATE,
    nationality VARCHAR(20),
    add_date DATE,
    update_date DATE
);

-- Create Movies Table
-- Create FK movies > directors
CREATE TABLE movies (
    movie_id SERIAL PRIMARY KEY,
    movie_name VARCHAR(100) NOT NULL,
    movie_length INT,
    date_of_birth DATE,
    movie_lang VARCHAR(20),
    age_certificate VARCHAR(10),
    release_date DATE,
    director_id INT REFERENCES directors (director_id)
);
```

```
-- Create Movies_Revenues Table
-- Create FK movies > directors
CREATE TABLE movies_revenues (
    revenue_id SERIAL PRIMARY KEY,
    movie_id INT REFERENCES movies (movie_id),
    revenues_domestic NUMERIC (10,2),
    revenues_international NUMERIC (10,2)
);

-- Create a Junction Table
-- Movies > Actors
CREATE TABLE movies_actors (
    movie_id INT REFERENCES movies (movie_id),
    actor_id INT REFERENCES actors (actor_id),
    PRIMARY KEY (movie_id, actor_id)
);
```

1.5 Import Movies Database Table Data

```
\i C:/Users/ellio/Documents/CODING-LESSONS/15-Advanced_PostgreSQL/resources/movies_database/directors.sql  
INSERT 0 37
```

```
\i C:/Users/ellio/Documents/CODING-LESSONS/15-Advanced_PostgreSQL/resources/movies_database/actors.sql  
INSERT 0 147
```

```
\i C:/Users/ellio/Documents/CODING-LESSONS/15-Advanced_PostgreSQL/resources/movies_database/movies.sql  
INSERT 0 53
```

```
\i C:/Users/ellio/Documents/CODING-LESSONS/15-Advanced_PostgreSQL/resources/movies_database/movies_actors.sql  
INSERT 0 70
```

```
\i C:/Users/ellio/Documents/CODING-LESSONS/15-Advanced_PostgreSQL/resources/movies_database/movies_revenues.sql  
INSERT 0 53
```

1.6 Creating databases & tables in PgAdmin 4

In PgAdmin 4, right click databases (in left pane) → Create database → in the general tab name the database “mydata”, select owner as postgres and save.

Right click on mydata → Create table → in the general tab name the table “accounts” → in the columns tab define as below and save.

Create - Table

General

Columns

Advanced

Constraints

Partitions

Parameters

Security

SQL

Inherited from table(s)

Select to inherit from...

Columns

	Name	Data type	Length/Precision	Scale	Not NULL?	Primary key?	Default
 	account_id	integer			☒	☒	nextval('a
 	username	character varying	100		☐	☐	
 	userpassword	character varying	50		☐	☐	
 	use_email	character varying	255		☐	☐	
 	add_date	time with time zon			☐	☐	
 	updated_date	time with time zon			☐	☐	

i?

Close

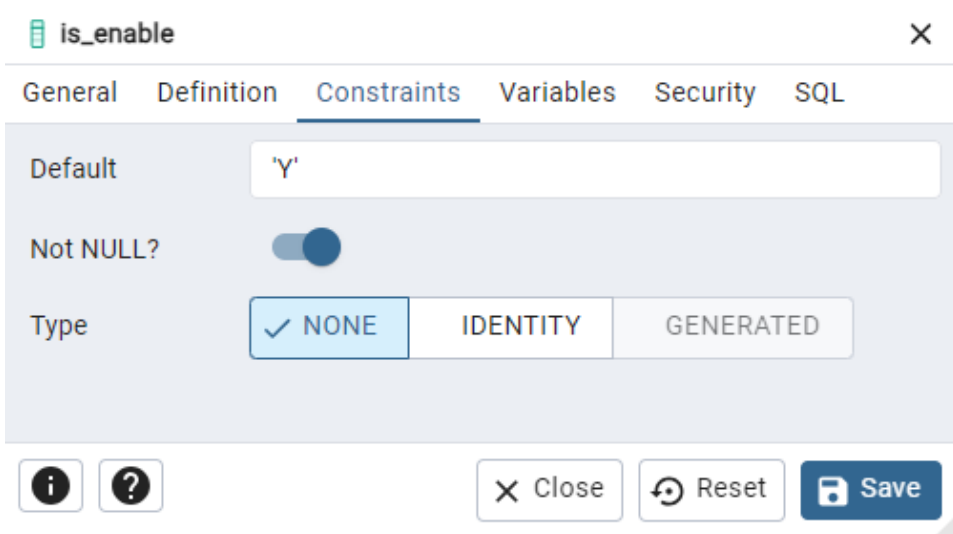
Reset

Save

1.7 Altering a Table in PgAdmin 4

In PgAdmin 4 under database 'mydata', expand tables and right click on accounts table → Select Create → Column. Add a new column with name "is_enabled" and save.

Expand the columns drop down in the right pane and right click on "is_enabled" → properties → In the constraints window define the default value as Y. Note that this has to be in single quotes. Set it to NOT NULL also. In the SQL tab it will show the SQL that will be executed to modify this table column. Click on Save.



The image shows the 'Constraints' tab for the column 'is_enable' in PgAdmin 4. The 'Default' field is set to 'Y'. The 'Not NULL?' toggle is turned on. The 'Type' is set to 'NONE'. The window has tabs for General, Definition, Constraints, Variables, Security, and SQL. At the bottom, there are buttons for Close, Reset, and Save.

Property	Value
Default	'Y'
Not NULL?	☒
Type	☒ NONE ☐ IDENTITY ☐ GENERATED

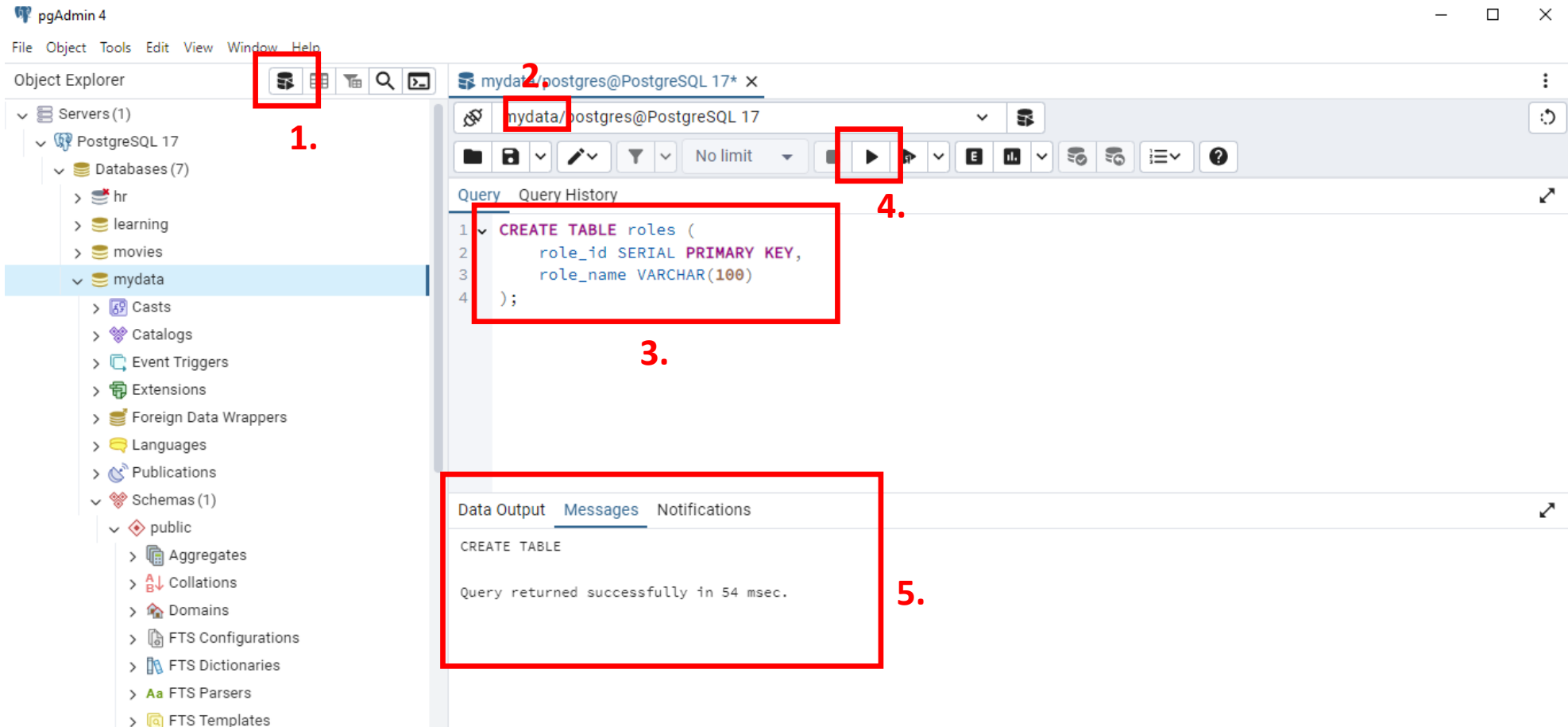


The image shows the 'SQL' tab for the column 'is_enable' in PgAdmin 4. It displays the SQL commands that will be executed to alter the column. The commands are: `ALTER TABLE IF EXISTS public.accounts ALTER COLUMN is_enable SET DEFAULT 'Y';` and `ALTER TABLE IF EXISTS public.accounts ALTER COLUMN is_enable SET NOT NULL;`. The window has tabs for General, Definition, Constraints, Variables, Security, and SQL. At the bottom, there are buttons for Close, Reset, and Save.

```
1 ALTER TABLE IF EXISTS public.accounts
2     ALTER COLUMN is_enable SET DEFAULT 'Y';
3
4 ALTER TABLE IF EXISTS public.accounts
5     ALTER COLUMN is_enable SET NOT NULL;
```

1.8 Using PgAdmin Query Tool

1. With the correct database selected, click the query tool button
2. Verify that the query tool has opened for the correct database
3. Write the SQL query as normal
4. Click the Query run button.
5. The result of the query shows in the bottom pane



2. Modifying Data in the Tables

[2.1 Create Tables using pgAdmin Query tool](#)

[2.2 Results panel of PgAdmin Query Tool](#)

[2.3 SQL Inserting multiple rows & Escaping apostrophe in a string](#)

[2.4 Returning Clause on an Insert Query](#)

[2.5 Update Single & Multiple Columns, Return Row](#)

[2.6 Delete Data & Upsert](#)

2.1 Create Tables using pgAdmin Query tool

In the Movies database create a customer table using pgAdmin Query tool.

The screenshot displays the pgAdmin 4 web interface. On the left, the 'Object Explorer' shows a tree structure with 'Servers' > 'PostgreSQL 17' > 'Databases (7)'. The 'movies' database is selected and highlighted. The main pane shows the 'Query' editor for the 'movies/postgres@PostgreSQL 17' connection. The SQL query is as follows:

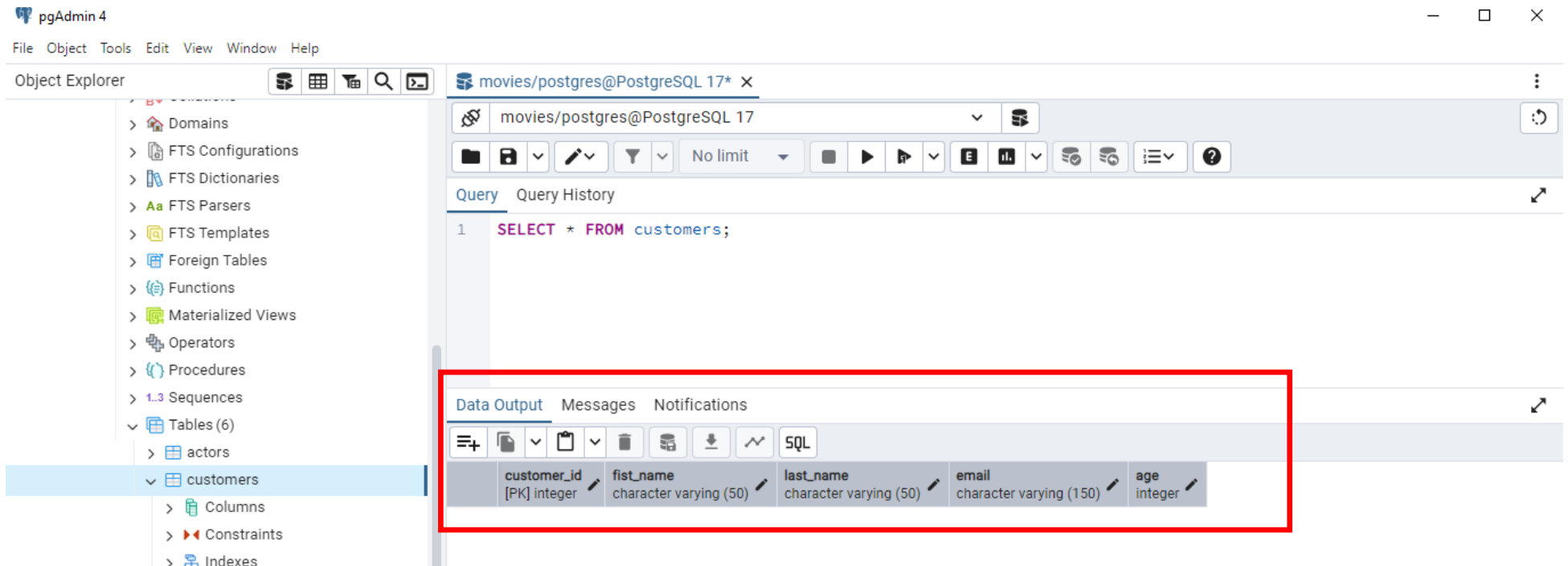
```
1 CREATE TABLE customers (  
2     customer_id SERIAL PRIMARY KEY,  
3     first_name VARCHAR(50),  
4     last_name VARCHAR(50),  
5     email VARCHAR (150),  
6     age int  
7 );
```

Below the query editor, the 'Messages' tab shows the execution result: 'CREATE TABLE' and 'Query returned successfully in 59 msec.'.

On the right side of the interface, there is a dark-themed code block containing the following SQL code:

```
-- Movies database customer table  
CREATE TABLE customers (  
    customer_id SERIAL PRIMARY KEY,  
    first_name VARCHAR(50),  
    last_name VARCHAR(50),  
    email VARCHAR (150),  
    age int  
);
```

2.2 Results panel of PgAdmin Query Tool



PgAdmin4 tool gives a table output for select queries.

2.3 SQL Inserting multiple rows & Escaping apostrophe in a string

Insert a row into a table

```
INSERT INTO customers (first_name, last_name, email, age)
VALUES ('Adnan', 'Waheed', 'a@b.com', 40);
```

Insert multiple rows into a table

```
INSERT INTO customers (first_name, last_name)
VALUES
('ADNAN', 'WAHEED'),
('JOHN', 'ADAMS'),
('LINDA', 'ABE');
```

Escape an apostrophe in a string
by preceding it with another
apostrophe

```
INSERT INTO customers (first_name, last_name)
VALUES ('Bill', 'O'Sullivan');
```

customer_id [PK] integer	first_name character varying (50)	last_name character varying (50)	email character varying (150)	age integer
1	Adnan	Waheed	a@b.com	40
2	ADNAN	WAHEED	[null]	[null]
3	JOHN	ADAMS	[null]	[null]
4	LINDA	ABE	[null]	[null]
7	Bill	O'Sullivan	[null]	[null]

2.4 Returning Clause on an Insert Query

Default insert query response

```
INSERT INTO customers (first_name)
VALUES ('ADAM');
INSERT 0 1
```

Insert query and return all columns of new row

```
INSERT INTO customers (first_name)
VALUES ('JOSEPH') RETURNING *;
```

	customer_id [PK] integer	first_name character varying (50)	last_name character varying (50)	email character varying (150)	age integer
1	9	JOSEPH	[null]	[null]	[null]

Insert query and return a specific column

```
INSERT INTO customers (first_name)
VALUES ('JOSEPH1') RETURNING
customer_id;
```

	customer_id [PK] integer
1	10

2.5 Update single & multiple columns, return Row

Update one column of a row

```
UPDATE customers
SET email = 'a2@b.com'
WHERE customer_id = 1;
```

Update multiple column of a row

```
UPDATE customers
SET email = 'a2@b.com', age = 30
WHERE customer_id = 1;
```

Update row and return row

```
UPDATE customers
SET email = 'a2@b.com'
WHERE customer_id = 1
RETURNING *;
```

	customer_id [PK] integer	first_name character varying (50)	last_name character varying (50)	email character varying (150)	age integer
1	1	Adnan	Waheed	a2@b.com	30

Add new column to table

```
ALTER TABLE IF EXISTS public.customers
ADD COLUMN is_enable VARCHAR(2);
```

Update a column for all rows

```
UPDATE customers
SET is_enable = 1;
```

2.6 Delete Data & Upsert

Delete a specific row of data

```
DELETE FROM customers  
WHERE customer_id = 9;
```

Delete **ALL** rows in a table

```
DELETE FROM customers;
```

Create new table in movies database

```
CREATE TABLE t_tags(  
  id serial PRIMARY KEY,  
  tag text UNIQUE,  
  update_date TIMESTAMP DEFAULT NOW()  
);
```

Insert some test data

```
INSERT INTO t_tags (tag) values  
('Pen'),  
('Pencil');
```

On conflict do nothing

```
INSERT INTO t_tags (tag)  
VALUES ('Pen')  
ON CONFLICT (tag) DO NOTHING;  
INSERT 0 0
```

On conflict set new values

```
INSERT INTO t_tags (tag)  
VALUES ('Pen')  
ON CONFLICT (tag)  
DO  
  UPDATE SET  
    tag = EXCLUDED.tag || '1',  
    update_date = NOW();
```

3. Querying Data from a Database

[3.1 Selecting data from a Table & Column Aliases](#)

[3.2 Select Query with Expressions](#)

[3.3 Sorting with Order By](#)

[3.4 Order By Alias & Expression & Column Number](#)

[3.5 Order by with Null Values](#)

[3.6 Using Distinct](#)

3.1 Selecting data from a Table & Column Aliases

Select all rows and all columns from movies table. Note that SQL will evaluate the FROM clause before the SELECT clause. Using * (all) in very large tables may affect server and application performance.

```
SELECT * FROM movies;
```

Select specific columns from movies table. Better because selecting specific columns reduces server load.

```
SELECT movie_name, movie_length
FROM movies;
```

Note that when I alias a column name PostgreSQL will convert all letters in the aliased column name to lowercase. Note how actor_firstName becomes actor_firstname.

```
SELECT
first_name AS actor_firstName,
last_name AS actor_lastName
FROM actors;
```

	actor_firstname character varying (150)	actor_lastname character varying (150)
1	Malin	Akerman
2	Tim	Allen
3	Julie	Andrews

If I want human readable aliases that contain forbidden characters such as spaces and capital letters then I can wrap the alias name in double quotes. Single quotes will not work.

```
SELECT
movie_name "Movie",
movie_lang "Language"
FROM movies;
```

The AS keyword is not needed if using double quotes when aliasing.

```
SELECT
first_name AS "Actor First Name",
last_name AS "Actor Last Name"
FROM actors;
```

	Actor First Name character varying (150)	Actor Last Name character varying (150)
1	Malin	Akerman
2	Tim	Allen
3	Julie	Andrews

3.2 Select Query with Expressions

We can combine two or more columns into one using concatenation (||). Note how SQL does not know how to name this column so it appears as “?column?”

```
SELECT  
first_name || ' ' || last_name  
FROM actors;
```

	?column? text
1	Malin Akerman
2	Tim Allen
3	Julie Andrews

WE can alias concatenated columns. Note how it appears as a datatype of TEXT despite both source columns being VARCHAR.

```
SELECT  
first_name || ' ' || last_name AS "Actor Name"  
FROM actors;
```

	Actor Name text
1	Malin Akerman
2	Tim Allen
3	Julie Andrews

3.3 Sorting with Order By

Default is to order the sort by ascending

```
SELECT * FROM movies
ORDER BY release_date;
```

First the FROM will be executed to find the table then the Columns will be identified then the ordering will be performed.

The Ascending (ASC) can be specifically defined

```
SELECT * FROM movies
ORDER BY release_date ASC;
```

The Ascending (ASC) can be specifically defined

```
SELECT * FROM movies
ORDER BY release_date DESC;
```

The first column name will be sorted first then the second column.

```
SELECT * FROM movies
ORDER BY
movie_lang DESC,
release_date ASC;
```

	movie_id [PK] integer	movie_name character varying (100)	movie_length integer	movie_lang character varying (20)	age_certificate character varying (10)	release_date date	director_id integer
1	20	Let the Right One In	128	Swedish	15	2008-10-24	1
2	26	Pans Labyrinth	98	Spanish	PG	2006-12-29	7
3	6	City of God	145	Portuguese	18	2003-01-17	20
4	7	City of Men	140	Portuguese	15	2008-02-29	22
5	25	Oldboy	130	Korean	18	2005-03-25	23
6	3	Battle Royale	111	Japanese	18	2001-01-04	10
7	34	Spirited Away	120	Japanese	U	2001-06-19	21
8	27	Ponyo	107	Japanese	U	2009-08-14	21
9	8	Cold Fish	108	Japanese	18	2010-09-12	30

3.4 Order By Alias & Expression & Column Number

```
SELECT
first_name firstname,
last_name surname
FROM actors
ORDER BY last_name;
```

I am aliasing columns
but sorting by the
original column name.

	firstname character varying (150) 🔒	surname character varying (150) 🔒
1	Malin	Akerman
2	Tim	Allen
3	Julie	Andrews
4	Ivana	Baquero

```
SELECT
first_name firstname,
last_name surname
FROM actors
ORDER BY surname;
```

I can also
sort by
the alias

	firstname character varying (150) 🔒	surname character varying (150) 🔒
1	Malin	Akerman
2	Tim	Allen
3	Julie	Andrews
4	Ivana	Baquero

```
SELECT
first_name,
LENGTH(first_name),
last_name
FROM actors
ORDER BY LENGTH;
```

I can sort
by an
expression

	first_name character varying (150) 🔒	length integer 🔒	last_name character varying (150) 🔒
1	Ian	3	Holm
2	Val	3	Kilmer
3	Ika	3	Nord
4	Tom	3	Hanks

```
SELECT
first_name,
LENGTH(first_name),
last_name
FROM actors
ORDER BY 1;
```

I can sort by
column number.
Note first
column is 1

	first_name character varying (150) 🔒	length integer 🔒	last_name character varying (150) 🔒
1	Ian	3	Holm
2	Val	3	Kilmer
3	Ika	3	Nord

The column numbers are derived from the
order they are listed in the select query, not
the order they appear in the table.

3.5 Order by with Null Values

```
INSERT INTO customers(first_name, last_name, email, age, is_enable)
VALUES
('John', 'Smith', 'johnsmith@hotmail.com', 42, 1),
('Jane', 'Parker', 'janeparker@yahoo.com', 31, 1),
('Ann', 'Ymous', null, 3, 1);
```

Default behaviour is to sort with null values last.

```
SELECT * FROM customers
ORDER BY email;
```

	customer_id [PK] integer	first_name character varying (50)	last_name character varying (50)	email character varying (150)	age integer	is_enable character varying (2)
1	15	Jane	Parker	janeparker@yahoo.com	31	1
2	14	John	Smith	johnsmith@hotmail.com	42	1
3	16	Ann	Ymous	[null]	3	1

I can specify that I want the Nulls last.

```
SELECT * FROM customers
ORDER BY email NULLS LAST;
```

	customer_id [PK] integer	first_name character varying (50)	last_name character varying (50)	email character varying (150)	age integer	is_enable character varying (2)
1	15	Jane	Parker	janeparker@yahoo.com	31	1
2	14	John	Smith	johnsmith@hotmail.com	42	1
3	16	Ann	Ymous	[null]	3	1

Or nulls first

```
SELECT * FROM customers
ORDER BY email DESC NULLS FIRST;
```

	customer_id [PK] integer	first_name character varying (50)	last_name character varying (50)	email character varying (150)	age integer	is_enable character varying (2)
1	16	Ann	Ymous	[null]	3	1
2	14	John	Smith	johnsmith@hotmail.com	42	1
3	15	Jane	Parker	janeparker@yahoo.com	31	1

3.6 Using Distinct

The distinct keyword will pick the first occurrence of each distinct entry for the specified column or columns.

```
SELECT DISTINCT movie_lang
FROM movies
ORDER BY 1;
```

	movie_lang character varying (20) 🔒
1	Chinese
2	English
3	German
4	Japanese
5	Korean
6	Portuguese
7	Spanish
8	Swedish

```
SELECT DISTINCT movie_lang, director_id
FROM movies
ORDER BY 1;
```

Find all rows where two columns are distinct

	movie_lang character varying (20) 🔒	director_id integer 🔒
1	Chinese	15
2	Chinese	37
3	Chinese	16
4	Chinese	35
5	English	14
6	English	13

This tells us that there are 4 directors making movies in the Chinese language

Find all rows where all columns are unique

```
SELECT DISTINCT *
FROM movies
ORDER BY 1;
```

Data Output Messages Notifications

	movie_id [PK] integer	movie_name character varying (100)	movie_length integer	movie_lang character varying (20)	age_certificate character varying (10)	release_date date	director_id integer
1	1	A Clockwork Orange	112	English	18	1972-02-02	13
2	2	Apocalypse Now	168	English	15	1979-08-15	9
3	3	Battle Royale	111	Japanese	18	2001-01-04	10
4	4	Blade Runner	121	English	15	1982-06-25	27

4. Filtering Data

[4.1 Selecting data from a Table & Column Aliases](#)

[4.2 And Or Operator, Combining And with OR Operator](#)

[4.3 SQL Query Order of execution for AND and OR](#)

[4.4 Where Clause Order of Operation & Alias in Where Clause](#)

[4.5 Using Comparison Operators with numeric, dates, text](#)

[4.6 Using Limit & Offset](#)

[4.7 Using Fetch](#)

[4.8 Using IN & Not IN](#)

[4.9 Using BETWEEN & Not BETWEEN](#)

[4.10 Using LIKE & ILIKE](#)

[4.11 Using Is NULL & IS NOT NULL](#)

[4.12 Concatenation, CONCAT & CONCAT_WS](#)

4.1 Comparison, Logical & Arithmetic Operators

-- Comparison Operators

```
-----  
-- Equal to                =  
-- Greater Than            >  
-- Less than               <  
-- Greater than or equal to >=  
-- Equal to or less than   <=
```

-- Logical Operators

```
-----  
-- AND  
-- OR  
-- LIKE  
-- IN  
-- BETWEEN
```

-- Arithmetic Operators

```
-----  
-- addition                +  
-- subtraction            -  
-- multiplication          *  
-- division                /  
-- modulus                 %
```


4.2 And Or Operator, Combining And with OR Operator

```
SELECT *  
FROM MOVIES  
WHERE movie_lang = 'English';
```

```
SELECT *  
FROM MOVIES  
WHERE movie_lang = English;  
ERROR: column "English"  
does not exist
```

```
SELECT *  
FROM MOVIES  
WHERE movie_lang ="English";  
ERROR: column "English"  
does not exist
```

When querying to match on a string, the string must be surrounded by single quotes.

```
SELECT *  
FROM MOVIES  
WHERE movie_lang ='English'  
AND Age_certificate = '18';
```

Note how the age certificate value is in single quotes because it is stored in the database as varchar not a number type.

```
SELECT *  
FROM MOVIES  
WHERE movie_lang = 'English'  
OR movie_lang ='Chinese'
```

Grammatically speaking this query is finding all rows where language is either English or Chinese

```
SELECT *  
FROM MOVIES  
WHERE movie_lang = 'English'  
AND director_id = 8;
```

Find all English language movies by director 8

```
SELECT *  
FROM MOVIES  
WHERE (movie_lang = 'English'  
OR movie_lang ='Chinese')  
AND Age_certificate = '12';
```

Get all English or Chinese movies that are age certification 12. Note that I have to wrap the movie language part in parenthesis to allow it to run the second condition.

4.3 SQL Query Order of execution for AND and OR

```
SELECT *  
FROM MOVIES  
WHERE movie_lang = 'English';
```

```
SELECT *  
WHERE movie_lang = English  
FROM MOVIES;  
ERROR: syntax error at or  
near "FROM"
```

The WHERE clause cannot come before the FROM clause. Order of operation is important.

```
SELECT *  
FROM MOVIES  
WHERE (movie_lang = 'English'  
OR movie_lang = 'Chinese')  
ORDER BY movie_lang;
```

```
SELECT *  
FROM MOVIES  
ORDER BY movie_lang;  
WHERE (movie_lang = 'English'  
OR movie_lang = 'Chinese')  
ERROR: syntax error at or  
near "WHERE"
```

The ORDER BY clause must come after the conditionals. This is because of the order of operation that SQL will execute a query

AND operator is processed first followed by the OR operator second.

SQL processes the AND operator like multiplication and the OR operator like addition, unless you include parenthesis.

Without parenthesis, the AND will be processed in the same way that $3*2+1$ would equal 7 but $3*(2+1)$ would equal 9.

It is better to include parenthesis whenever you are using multiple operators so that it is clear to you.

4.4 Where Clause Order of Operation & Alias in Where Clause

```
SELECT
first_name,
last_name AS surname
FROM actors
WHERE surname = 'Allen';
ERROR: column "surname"
does not exist
```

If you use an alias in a select query then you cannot use it in the WHERE clause of the same query. You can use it in a sub query though. More on sub queries later.

```
SELECT *
FROM movies
WHERE movie_lang = 'English'
ORDER BY movie_length;
```

PostgreSQL evaluates the WHERE clause after the FROM clause and before the SELECT and ORDER BY clause

FROM | WHERE | SELECT | ORDER BY

SQL works by first fetching the table data then it filters by the where clause and then selects the columns to display, finally performing the ordering.

4.5 Using Comparison Operators with numeric, dates, text

```
SELECT *  
FROM movies  
WHERE movie_length > 100  
ORDER BY movie_length;
```

```
SELECT *  
FROM movies  
WHERE movie_length >= 100  
ORDER BY movie_length;
```

Note that the equal symbol comes after the greater than or less than symbol.

```
SELECT *  
FROM movies  
WHERE movie_length < 100  
ORDER BY movie_length;
```

```
SELECT *  
FROM movies  
WHERE movie_length <= 100  
ORDER BY movie_length;
```

```
SELECT *  
FROM movies  
WHERE release_date >= '2000-01-01'  
ORDER BY release_date;
```

When querying for dates, it is important to first take a look at how the date is stored in the table you are querying. i.e. YYYY-MM-DD or DD-MM-YYYY etc. YYYY-MM-DD is the default.

```
SELECT *  
FROM movies  
WHERE movie_lang > 'English'  
ORDER BY movie_lang;
```

When querying text fields it does it alphabetically, i.e. anything after the letter E is found.

```
SELECT *  
FROM movies  
WHERE movie_lang != 'English'  
ORDER BY movie_lang;
```

!= is the NOT EQUAL TO operator

4.6 Using Limit & Offset

```
SELECT *  
FROM movies  
ORDER BY movie_name;  
LIMIT 10;
```

Limit the
number of
rows in SQL
query results

```
SELECT *  
FROM movies  
ORDER BY movie_length  
LIMIT 5;
```

```
SELECT *  
FROM movies_revenues  
ORDER BY revenues_domestic DESC NULLS LAST  
LIMIT 10;
```

Get the top 10 most domestic profitable movies

```
SELECT *  
FROM movies_revenues  
ORDER BY revenues_domestic ASC NULLS LAST  
LIMIT 10;
```

Get the least profitable domestic movies

```
SELECT *  
FROM movies_revenues  
ORDER BY revenues_domestic DESC NULLS LAST  
LIMIT 5  
OFFSET 5;
```

Get the most profitable domestic movies after the top 5. Note that using offset 5 means the results will start from row 6.

```
SELECT *  
FROM directors  
WHERE nationality = 'American'  
ORDER BY date_of_birth  
LIMIT 5;
```

Get the top 5
oldest American
directors

```
SELECT *  
FROM actors  
WHERE gender = 'F'  
ORDER BY date_of_birth  
LIMIT 5;
```

Get the 5
youngest female
actors

4.7 Using Fetch

Fetch clause is used to retrieve a portion of rows returned by a query

Fetch was introduced into SQL in 2008

Fetch clause is functionally equivalent to the LIMIT clause

Fetch clause is functionally equivalent to the LIMIT clause

Fetch syntax is:

OFFSET start (row | rows)

FETCH (FIRST | NEXT) [row_count] (row | rows) ONLY

```
SELECT *  
FROM movies  
FETCH FIRST ROW ONLY;
```

Get the first movie in the table

```
SELECT *  
FROM movies  
ORDER BY movie_length DESC  
FETCH FIRST 5 ROW ONLY;
```

Get the top 5 longest movies

```
SELECT *  
FROM movies  
FETCH FIRST 5 ROW ONLY;  
ORDER BY movie_length DESC  
ERROR: syntax error at or near "ORDER"
```

FETCH must come after ORDER

```
SELECT *  
FROM directors  
WHERE nationality = 'American'  
ORDER BY date_of_birth  
FETCH FIRST 5 ROW ONLY;
```

Get the top 5 oldest American directors

```
SELECT *  
FROM movies  
ORDER BY movie_length DESC  
OFFSET 5  
FETCH FIRST 5 ROW ONLY;
```

Get the 5 longest movies after the first five longest movies. **Offset works before or after the fetch.**

```
SELECT *  
FROM actors  
WHERE gender = 'F'  
ORDER BY date_of_birth DESC  
FETCH FIRST 10 ROW ONLY;
```

Get the 10 youngest female actors

```
SELECT *  
FROM movies  
ORDER BY movie_length DESC  
FETCH FIRST 5 ROW ONLY  
OFFSET 5;
```

4.8 Using IN & Not IN

IN & NOT IN are used to check if a value matches or not matches in a list and returns true or false

IN syntax is:

value IN (value1, value2, ...) or value NOT IN (value1, value2, ...)

```
SELECT *  
FROM movies  
WHERE movie_lang = 'English'  
OR movie_lang = 'Chinese'  
OR movie_lang = 'Japanese'  
ORDER BY movie_lang;
```

Get all movies for English, Chinese, and Japanese languages

```
SELECT *  
FROM movies  
WHERE movie_lang IN ('English', 'Chinese', 'Japanese')  
ORDER BY movie_lang;
```

An in clause works by looking at the array (or list of values and then matches true or false on the column specified. This is a much more scalable solution than writing multiple or clauses.

```
SELECT *  
FROM movies  
WHERE Age_certificate IN ('13', 'PG')  
ORDER BY Age_certificate;
```

Get all 13 and PG age certificate movies

```
SELECT *  
FROM movies  
WHERE director_id NOT IN (13, 10)  
ORDER BY director_id;
```

Get all movies where director_id is not 13 or 18. Note that director_id is data type integer so quotes are not needed.

```
SELECT *  
FROM actors  
WHERE actor_id NOT IN (1,2,3,4)  
ORDER BY actor_id;
```

Get all actors where actor_id is not 1,2,3 or 4

4.9 Using BETWEEN & Not BETWEEN

An operator to match a value against a range of values where Value is BETWEEN low value and high value. If the value is greater than or equal to the low value and less than or equal to the high value then it returns true, otherwise false.

```
SELECT *  
FROM actors  
WHERE date_of_birth >= '1991-01-01'  
AND date_of_birth <= '1995-12-31'  
ORDER BY date_of_birth;
```

Get all the actors where birth date is between 1991 and 1995

```
SELECT *  
FROM actors  
WHERE date_of_birth BETWEEN '1991-01-01' AND '1995-12-31'  
ORDER BY date_of_birth;
```

Get all the movies release
between 1998 and 2002

```
SELECT *  
FROM movies  
WHERE release_date BETWEEN '1998-01-01' AND '2002-12-31'  
ORDER BY release_date;
```

Get all movies with domestic
revenues between 100 and 300

```
SELECT *  
FROM movies_revenues  
WHERE revenues_domestic BETWEEN 100 AND 300  
ORDER BY revenues_domestic;
```

Note that numeric data type, i.e. a decimal
value (eg 102.10) can be used. Also between
matches on equal to or greater/less than.

```
SELECT *  
FROM movies_revenues  
WHERE revenues_domestic BETWEEN 102.10 AND 300  
ORDER BY revenues_domestic;
```

```
SELECT *  
FROM movies  
WHERE movie_length NOT BETWEEN 100 AND 200  
ORDER BY movie_length;
```

Get short and long movies that are not
between 100 and 200 minutes.

4.10 Using LIKE & ILIKE

Operators to query data using "pattern matching" that returns true or false. Both allow search for patterns in strings by using two special characters. % matches any sequence of zero or more characters, underscore matches any single character.

```
SELECT *  
FROM actors  
WHERE first_name LIKE 'A%'  
ORDER by first_name;
```

Get all actors
who's first name
starts with 'A'

```
SELECT *  
FROM actors  
WHERE last_name LIKE '%a'  
ORDER by last_name;
```

Get all actors
who's last name
ends with 'a'

```
SELECT *  
FROM actors  
WHERE first_name LIKE '_____  
ORDER by first_name;
```

Get all actors
whose first
name is five
characters

```
SELECT *  
FROM actors  
WHERE first_name LIKE '_l%'  
ORDER by first_name;
```

Get all actors
whose first name
contains a l as the
second character

```
SELECT *  
FROM actors  
WHERE first_name LIKE 'Tim%'  
ORDER by first_name;
```

Get all actors
who's first
name is Tim

```
SELECT *  
FROM actors  
WHERE first_name ILIKE 'tim%'  
ORDER by first_name;
```

Like is case
sensitive. Ilike
is not case
sensitive

4.11 Using Is NULL & IS NOT NULL

Is null operator checks if a value is null. IS NOT NULL checks if a value is not null. Null means that information is missing or not applicable. returns true or false.

```
SELECT *  
FROM actors  
WHERE date_of_birth IS NULL
```

```
SELECT *  
FROM actors  
WHERE date_of_birth = NULL;
```

```
SELECT *  
FROM actors  
WHERE date_of_birth = 'NULL';  
ERROR: invalid input syntax  
for type date: "NULL"
```

Find list of actors with missing birth date. Equal to Null will always return zero rows even if there are nulls. Null in quotes fir date and time types will fail. **ALWAYS USE IS NULL**

```
SELECT *  
FROM actors  
WHERE date_of_birth IS  
NULL  
OR first_name IS NULL  
ORDER by first_name;
```

Find list of actors
with missing birth
date OR missing
first name

```
SELECT *  
FROM movies_revenues  
WHERE revenues_domestic IS NULL  
OR revenues_international IS  
NULL;
```

Find movies
where either
domestic or
international
revenues are null

```
SELECT *  
FROM movies_revenues  
WHERE revenues_domestic IS NULL
```

Find list of movies where
domestic revenues are null

```
SELECT *  
FROM movies_revenues  
WHERE revenues_domestic IS NOT NULL;
```

Find movies where domestic revenue
is not null

4.12 Concatenation, CONCAT & CONCAT_WS

in SQL language double pipe (||) means concatenate, not OR.

```
SELECT  
first_name || ' ' || last_name AS fullname  
FROM actors;
```

The double pipe is used to make a concatenated column. Note how when using the double pipe I also need to add in the space character between first name and last name

```
SELECT  
CONCAT (first_name, last_name) AS fullname  
FROM actors;
```

```
SELECT  
CONCAT (first_name, ' ', last_name) AS fullname  
FROM actors;
```

The concat function takes an array of columns to concatenate but performs the concatenation without spaces between the strings. I can manually add in a space.

```
SELECT  
CONCAT_WS ('', first_name, last_name, date_of_birth) AS actordata  
FROM actors;
```

CONCAT_WS performs concatenation with a separator. Note that the separator is defined as the first item in the array of values.

```
SELECT 'hello' || null || 'world';
```

The expected result of this would be “helloworld” but in fact a null in a double pipe concatenation returns null.

```
SELECT 'hello' || 'null' || 'world';
```

Null inside quotes is treated as a sting and will return “hellonullWorld”

```
SELECT
revenues_domestic,
revenues_international,
CONCAT(revenues_domestic, revenues_international) AS profits
FROM movies_revenues;
```

```
SELECT
revenues_domestic,
revenues_international,
CONCAT_WS('|',revenues_domestic, revenues_international) AS profits
FROM movies_revenues;
```

	revenues_domestic numeric (10,2) 	revenues_international numeric (10,2) 	profits text 
2	199.40	201.20	199.40201.20
3	102.10	[null]	102.10
4	158.70	[null]	158.70
5	27.10	[null]	27.10
6	[null]	[null]	
7	260.30	210.90	260.30210.90
8	28.10	[null]	28.10

CONCAT and CONCAT_WS functions will return a value if one of the columns contains a value but will return NULL IF ALL COLUMNS ARE NULL because it is smart function that ignores nulls

5. PostgreSQL Data Types

[5.1 Boolean](#)

[5.2 CHAR, VARCHAR & Text](#)

[5.3 Numeric Data Types](#)

[5.4 Date and Time Data Types](#)

[5.5 UUID Data Type](#)

[5.6 Array Data Type](#)

[5.6 Array Data Type](#)

[5.7 JSON](#)

[5.8 Network Addresses](#)

5.1 Boolean

Boolean is a datatype with three states:
True | False | Null

Some valid literals for Boolean values in PostGreSQL are:

Note that they must be enclosed in single quotes except TRUE or FALSE

```
-- TRUE      FALSE
-- ----      -
-- TRUE      FALSE
-- 'true'    'false'
-- 't'       'f'
-- 'y'       'n'
-- 'yes'     'no'
-- '1'       '0'
```

```
CREATE TABLE table_boolean (
  product_id SERIAL PRIMARY KEY,
  is_available BOOLEAN NOT NULL
)
```

In the learning database create the table

Columns									+
		Name	Data type	Length/Precision	Scale	Not NULL?	Primary key?	Default	
		product_id	integer v			☒	☒	nextval('t	
		is_available	boolean v			☒	☐		

```
INSERT INTO table_boolean (is_available) VALUES (TRUE);
INSERT INTO table_boolean (is_available) VALUES (FALSE);
INSERT INTO table_boolean (is_available) VALUES ('true');
INSERT INTO table_boolean (is_available) VALUES ('0');
INSERT INTO table_boolean (is_available) VALUES ('t');
INSERT INTO table_boolean (is_available) VALUES ('no');
```

	product_id [PK] integer	is_available boolean
1	1	true
2	2	false
3	3	true
4	4	false
5	5	true
6	6	false

```
SELECT *
FROM table_boolean
WHERE is_available = '1';
```

	product_id [PK] integer	is_available boolean
1	1	true
2	3	true
3	5	true

```
SELECT *
FROM table_boolean
WHERE is_available = 't';
```

	product_id [PK] integer	is_available boolean
1	1	true
2	3	true
3	5	true

```
SELECT *
FROM table_boolean
WHERE is_available = FALSE;
```

	product_id [PK] integer	is_available boolean
1	2	false
2	4	false
3	6	false

```
-- TRUE FALSE
-- ----
-- TRUE FALSE
-- 'true' 'false'
-- 't' 'f'
-- 'y' 'n'
-- 'yes' 'no'
-- '1' '0'
```

Any of the permitted true or false types can be used in a where clause

```
SELECT *
FROM table_boolean
WHERE is_available;
```

	product_id [PK] integer	is_available boolean
1	2	false
2	4	false
3	6	false

```
SELECT *
FROM table_boolean
WHERE is_available;
```

	product_id [PK] integer	is_available boolean
1	1	true
2	3	true
3	5	true

I do not need an equals clause on Boolean datatypes.

```
ALTER TABLE table_boolean
ALTER COLUMN is_available
SET DEFAULT FALSE;
```

```
ALTER TABLE table_boolean
ALTER COLUMN is_available
SET DEFAULT '0';
```

```
ALTER TABLE table_boolean
ALTER COLUMN is_available
SET DEFAULT 'false';
```

I can set a default value for Boolean

is_available

General Definition Constraints Variables Security SQL

Default

Not NULL? ☒

Type ✓ NONE IDENTITY GENERATED

```
INSERT INTO table_boolean
(product_id) VALUES (7);
```

	product_id [PK] integer	is_available boolean
1	1	true
2	2	false
3	3	true
4	4	false
5	5	true
6	6	false
7	7	false

5.2 CHAR, VARCHAR & Text

Character string types are general purpose types suitable for:
text | numbers | symbols

```
-- CHARACTER(n)           Fixed length Blank padded
-- CHAR(n)                Fixed length Blank padded
--
-- CHARACTER VARYING(n)    Variable length with length limit
-- VARCHAR(N)             Variable length with length limit
```

5.2a CHAR

N is the number of characters and if n is not specified then it defaults to 1. If the excessive characters are all spaces, PostgreSQL truncates the spaces to the maximum length and stores the characters.

CHAR(10) will store 10 character length. However, if the data inserted is less than 10 characters, PostgreSQL will pad the data with spaces to make up the length 10.

```
SELECT CAST ('Elliott' AS CHARACTER(10)) AS "name";
SELECT 'Elliott'::CHAR(10) AS "name";
```

	name character (10)
1	Elliott

"Elliott "

I can CAST a string to a particular data type

```
SELECT 'Elliott'::CHAR AS "name";
```

	name character (1)
1	E

Where length is not specified, the default is 1

CHAR can be used where the datatype is text of a known fixed length such as a two character country code or a three character city code, airport codes.

5.2a VARCHAR

VARCHAR is useful if entries have a varying length and padding is not needed to fill with blanks.

VARCHAR stores exactly the number of characters provided and saves space by not padding with blank spaces. No default value exists for this type. n means the maximum number of characters permitted.

```
SELECT CAST ('Elliott' AS VARCHAR(10)) AS "name";  
SELECT 'Elliott'::VARCHAR(10) AS "name";
```

name character (10) 🔒	
1	Elliott

“Elliott”

```
SELECT 'THIS IS A TEST'::VARCHAR(10) AS "name";
```

name character varying (10) 🔒	
1	THIS IS A

If the characters exceeds the specified length they will be truncated.

```
SELECT 'AZ 19 %$'::VARCHAR(10) AS "name";
```

name character varying (10) 🔒	
1	AZ 19 %\$

Varchar can store Text, Numbers and symbols

5.2c Text

TEXT is a variable length column of any size unlimited length (postgreSQL specs say up to 1GB).

TEXT is not part of SQL standard but similar types are available in Microsoft SQL, MySQL, MariaDB

```
SELECT
'Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has
been the industry's standard dummy text ever since the 1500s, when an unknown printer took a
galley of type and scrambled it to make a type specimen book.'
::Text AS "text";
```

	text text
1	Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown

```
CREATE TABLE table_characters (
  col_char CHAR(10),
  col_varchar VARCHAR(10),
  col_text TEXT
)
```

```
INSERT INTO table_characters
(col_char, col_varchar, col_text)
VALUES
('ABC', 'ABC', 'ABC'),
('xyz', 'xyz', 'xyz');
```

```
-- CHAR          VARCHAR    TEXT
-- ----          -
-- "ABC          "  "ABC"      "ABC"
-- "xyz          "  "xyz"      "xyz"
```

Typically, using VARCHAR is good for small to large databases, but your choice of a good character data type should be based on:

1. Database design
2. Read/write frequency
3. Overall business logics

5.3 Numeric Data Types

Numbers columns can hold various type numbers but not Null values.

Math operators (addition, multiplication etc..) can be performed on numbers data type

```
-- INTEGERS      Whole numbers can be +ve or -ve
-- FLOATING POINT Varying decimal length for fraction
-- FIXED POINT    Fixed decimal length for fraction
```

5.3a INTEGER

```
-- INTEGER TYPES
-----
-- SMALLINT      2 BYTES  -32768                to +32767
-- INT           4 BYTES  -2147483648            to +2147483647
-- BIGINT        8 BYTES  -9223372036854775808    to +9223372036854775807

-- SMALLSERIAL   2 BYTES      Serial is an ANSI SQL
-- SERIAL        4 BYTES      standard for identity columns
-- BIGSERIAL     8 BYTES      that auto-increments
```

One of the three integer types will be good enough for any number situation.

Each time you use a bigint it will occupy 8-bytes of database storage space.

Bigint is good for up to 2.1 billion. Anything outside of this range and the database will give an error.

Smart database design is to use an integer type that will fit all possible data values and occupy the least bytes to save database storage space.

```
CREATE TABLE table_serial (
    product_id SERIAL,
    product_name VARCHAR(100)
);
```

```
INSERT INTO table_serial (product_name) VALUES ('Pen');
INSERT INTO table_serial (product_name) VALUES ('Pencil');
SELECT * FROM table_serial;
```

If I look at the **serial data type** it is an integer with an auto-increment constraint called Nextval. The Nextval is responsible for handling the auto-increment.

	product_id integer	product_name character varying (100)
1	1	Pen
2	2	Pencil

product_id

General Definition Constraints Variables Security SQL

Data type integer

Length/Precision

Scale

product_id

General Definition Constraints Variables Security SQL

Default nextval('table_boolean_product_id_seq'::regclass)

Not NULL? ☒

Type ✓ NONE IDENTITY GENERATED

Because auto-increment adds 1 to the last entry, if a row is deleted it does not go back and re-use that vacated value.

5.3b DECIMALS – Fixed Point

Decimals represent a whole number plus a fraction of a whole number

The fraction is represented by digits following a decimal point

```
-- Numeric(precision, scale)
-----
-- Precision    Maximum number of digits to the left and right of the decimal point
-- Scale        Number of digits allowable on the right of the decimal point
```

`numeric(10,2)` Will return 10 digits with two of those digits being to the right of the decimal point. `decimal(10,2)` point. Numeric and decimal are functionally the same in PostgreSQL.

5.3c DECIMALS – Floating Point

Unlike numeric, where we specifically specify fixed precision and scale (i.e. `numeric(10,2)`), the decimal of a given column can float depending on the number.

```
-- Float
-----
-- real    allows precision to 6 decimal places
-- double  allows precision to 15 decimal places
```

Data Type	Storage Size	Storage Type	Range
Numeric, Decimal	Variable	Fixed Point	up to 131072 digits before the decimal point Upton 17383 digits after the decimal point
real	4 bytes	Floating Point	6 decimal digits precision
Double Precision	8 bytes	Floating Point	15 decimal digits precision

5.3d Default behaviours of Numeric, Real & Double Precision

```
CREATE TABLE table_numbers (  
  col_numeric NUMERIC(20,5),  
  col_real REAL,  
  col_double DOUBLE PRECISION  
)
```

```
SELECT * FROM table_numbers;
```

```
INSERT INTO table_numbers  
(col_numeric, col_real, col_double)  
VALUES  
(.9,.9,.9),  
(3.13579,3.13579,3.13579),  
(4.1357987654,4.1357987654,4.1357987654);
```

	col_numeric numeric (20,5) 🔒	col_real real 🔒	col_double double precision 🔒
1	1. 0.90000	2. 0.9	2. 0.9
2	3.13579	3.13579	3.13579
3	3. 4.13580	4. 4.135799	5. 4.1357987654

- 1.** The numeric data type is fixed length so 0.9 is padded out to 5 decimal places with zero's.
- 2.** Real and double precision data types do not pad out the decimal part with zeros.
- 3.** The numeric data point has a precision of 5 decimal places so it will truncate and round.
- 4.** The real datapoint is limited to a precision of six decimal places so data is truncated and rounded.
- 5.** Double precision stores up to 15 decimal digits so the whole number is stored.

5.3e Choosing the correct number data type

1. Use integers wherever possible.
2. Decimal data and calculations need to be exact, then use numeric or decimal. Float will save space but be careful about inexactness.
3. Chose a big enough number type by looking at your data low ranges and high ranges and the number of digits after the decimal point.
4. With big whole numbers, use bigint only if column values are constrained to fit into either the smaller int or smallint types. i.e a table that can have potentially millions of rows bigint would be used.

5.4 Date and Time Data Types

Date and time are the most important data types in any database structure.

PostgreSQL date & Time data types are:

Data Type	Stores	Low value	High Value
Date	Date only	4713 BC	294276 AD
Time	Time only	4713 BC	5874897 AD
Timestamp	Date & Time	4713 BC	294276 AD
Timestamptz	Date & Time & Timezone	4713 BC	294276 AD
Interval	Stores difference between two data types		

5.4a Data Data type

Stores Date values using 4 Bytes by default using format YYYY-MM-DD.
CURRENT_DATE is a useful keyword for storing current date.

```
CREATE TABLE table_date (  
  id SERIAL PRIMARY KEY,  
  employee_name VARCHAR(100) NOT NULL,  
  hire_date DATE NOT NULL,  
  add_date DATE DEFAULT CURRENT_DATE  
);
```

```
INSERT INTO table_date  
(employee_name, hire_date)  
VALUES  
  ('adam', '2020-01-01'),  
  ('linda', '2020-02-01');
```

```
SELECT *  
FROM table_date;
```

Note that add_date column has been automatically filled with a default date of the current date.

	id [PK] integer	employee_name character varying (100)	hire_date date	add_date date
1	1	adam	2020-01-01	2025-01-13
2	2	linda	2020-02-01	2025-01-13

```
SELECT CURRENT_DATE, NOW();
```

	current_date date	now timestamp with time zone
1	2025-01-13	2025-01-13 18:41:08.353094+01

Note that CURRENT_DATE only returns the date as YYYY-MM-DD whereas the NOW() function returns the Date (YYYY-MM-DD), Time (hh:mm:ss.ms) and timezone (UTC+1)

5.4b Time Data type

Time stores the time of the day value as approximately 8 bytes. Column_name TIME(precision) where precision is the number of fractional digits placed in the second field. precision up to 6 digits.

Time can be stored in many formats with different degrees of precision.

Common Formats	Description	Example	Not valid
HH:MM	Hours : minutes	10:30	
HH:MM:SS	Hours : minutes : seconds	23:59:25	24:60:60
HHMMSS	HoursMinutesSecods	103025	
MM:SS.pppppp	minutes : seconds . precision	03:59. 123456	
HH:MM:SS.pppppp	Hours : minutes : seconds . precision	10:03:59.123456	
HHMMSS.pppppp	HoursMinutesSecods . precision	103025. 123456	

```
CREATE TABLE table_time (  
  id SERIAL PRIMARY KEY,  
  class_name VARCHAR(100) NOT NULL,  
  start_time TIME NOT NULL,  
  end_time TIME NOT NULL  
);
```

```
INSERT INTO table_time  
(class_name, start_time, end_time)  
VALUES  
( 'MATHS', '08:00:00', '09:00:00'),  
( 'CHEMISTRY', '09:01', '10:00:00');
```

```
SELECT *  
FROM table_time;
```

	id [PK] integer	class_name character varying (100)	start_time time without time zone	end_time time without time zone
1	1	MATHS	08:00:00	09:00:00
2	2	CHEMISTRY	09:01:00	10:00:00

Note that when I insert time into the database I can specify the precision in one of the formats above.

```
SELECT CURRENT_TIME, CURRENT_TIME(3), LOCALTIME, LOCALTIME(3);
```

	current_time time with time zone 🔒	current_time time with time zone (3) 🔒	localtime time without time zone 🔒	localtime time without time zone (3) 🔒
1	19:19:29.973732+01:00	19:19:29.974000+01:00	19:19:29.973732	19:19:29.974

CURRENT_TIME is the server time and LOCALTIME is the time on the machine running the command. In this case as the server is running on the machine calling the time functions it is the same. Both these functions accept precision.

Note the three trailing zeros in the precision of CURRENT TIME and how the trailing zeros are omitted in the precision of LOCALTIME.

```
SELECT TIME '10:00' - TIME '04:00' AS result;
```

Arithmetic operations can be performed on time data type

	result interval 🔒
1	06:00:00

```
SELECT CURRENT_TIME + interval '2 hour' AS result
```

Interval 'n type' where n is a value and type is the units i.e. hours

	result time with time zone 🔒
1	21:34:47.853932+01:00

5.4b TIMESTAMP & TIMESTAMPTZ format

Allow storage of date and time together where **Timestamp is without timezone** and **timestamptz is with timezone**. The value of the timezone is always UTC.

An input value that has an **explicit timezone specified is converted to UTC** using the appropriate **offset for that timezone**. If not Timezone is stated in the input string then is is assumed to be in the **timezone of the server indicated by the system's timezone parameter** and is converted to UTC using the offset for the timezone.

When a timestampTZ value is output, it is **always converted from UTC to the current time zone, and displayed as local time in that zone**.

```
CREATE TABLE table_time_tz (  
  ts TIMESTAMP,  
  tstz TIMESTAMPTZ  
);
```

```
INSERT INTO table_time_tz (ts,tstz)  
VALUES ('2020-02-22 10:10:10-07', '2020-02-22 10:10:10-07');
```

```
SELECT * FROM table_time_tz;
```

	ts timestamp without time zone	tstz timestamp with time zone
1	2020-02-22 10:10:10 1.	2020-02-22 18:10:10+01 2.

```
SHOW TIMEZONE;
```

3.	TimeZone text
1	Europe/Paris

- 1.** Note how that even though in the ts column the time was 10:10:10-07 the timezone of UTC+07 hours was ommitted.
- 2.** In the tstz column note how even though the time with timezone was 10:10:10-07 the time is output in localtime of the machine running the query.
- 3.** It is possible to see the timezone that I am in with show timezone.

```
SET TIMEZONE = 'Europe/Paris';
```

```
SELECT * FROM table_time_tz;
```

1. If I set the TimeZone to my actual location then the TimestampTZ output will be the same as the values originally entered.

	ts timestamp without time zone	tstz timestamp with time zone
1	2020-02-22 10:10:10	2020-02-22 18:10:10+01 1.

```
SELECT  
CURRENT_TIMESTAMP,  
TIMEOFDAY();
```

	current_timestamp timestamp with time zone	timeofday text
1	2025-01-13 22:04:52.622823+01	Mon Jan 13 22:04:52.622989 2025 CET

Show current date & Time with Timezone and show date, time and timezone as a more human readable format.

```
SELECT TIMEZONE ('Asia/Singapore', '2025-01-01 00:00:00');
```

	timezone timestamp without time zone
1	2025-01-01 07:00:00

Timezone can be used to convert time from one timezone to another.

🏠	CET / CE... Central Europ...	12:00a - 12:30a Mon, Jan 13 Mon, Jan 13
-1	UTC Coordinated ...	11:00p - 11:30p Sun, Jan 12 Sun, Jan 12
+7	Singapor... Singapore	7:00a - 7:30a Mon, Jan 13 Mon, Jan 13

MON	JAN 13	1 am	2 am
1	JAN 13	1 am	2 am
7	8	9	

Note my current timezone is CET and in Singapore, midnight CET is 7am.

5.5 UUID Data Type

UUID is a 128 bit globally unique number generated by an algorithm. it is represented as a 32 digit HEX string separated by hyphens. An UUID algorithm is defined by a third party monitorium so you will need to add the UUID extension. See my Beginner PostgreSQL course for how to install the UUID extension and generate UUID numbers.

```
CREATE EXTENSION IF NOT EXISTS "uuid-oss";
```

Install the UUID extensions on the database.

```
CREATE TABLE table_uuid (  
  product_id UUID DEFAULT uuid_generate_v1(),  
  product_name VARCHAR(100) NOT NULL  
);
```

Create a table that will call v1 UUID to default assign a random number to product_id.

```
INSERT INTO table_uuid  
(product_name) VALUES ('pen');
```

```
INSERT INTO table_uuid  
(product_name) VALUES ('pencil');
```

```
SELECT * FROM table_uuid;
```

	product_id uuid	product_name character varying (100)
1	43a58316-d1f6-11ef-8452-231a238a9c03	pen
2	d2b7442c-d1f6-11ef-8453-dfd0af657a90	pencil

UUID generates a globally unique value. Note for each row that the Id values are completely different but based on MAC address, timestamp and a random component so there is some commonality 11ef_84.

```
ALTER TABLE table_uuid  
ALTER COLUMN product_id  
SET DEFAULT uuid_generate_v4();
```

I can modify the column to now generate a completely random number using UUID v4.

```
INSERT INTO table_uuid  
(product_name) VALUES ('rubber');
```

```
INSERT INTO table_uuid  
(product_name) VALUES ('eraser');
```

	product_id uuid	product_name character varying (100)
1	43a58316-d1f6-11ef-8452-231a238a9c03	pen
2	d2b7442c-d1f6-11ef-8453-dfd0af657a90	pencil
3	03f5577d-3fa8-4b3c-8e91-26512d7ee94f	rubber
4	fa687350-9e33-4298-b6f5-2d629361bdfb	eraser

5.6 Array Data Type

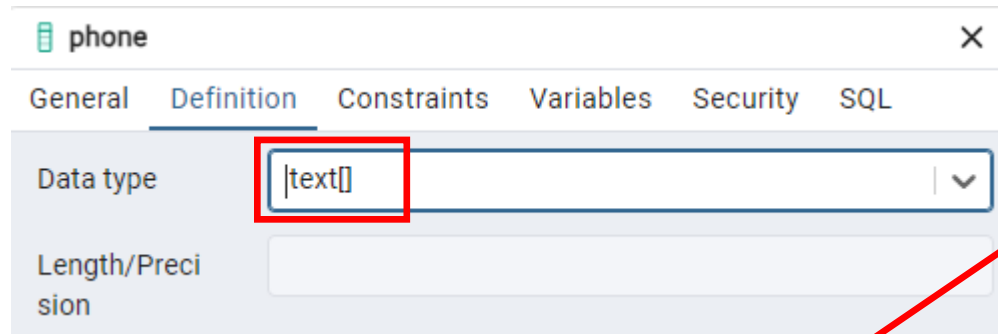
Every datatype has its own companion array data type. I.e. Integer has an integer[] array type, Character has character[] array type. Array datatype is named by appending square brackets ([]) to the data type name of the array elements.

```
CREATE TABLE
table_array (
  id SERIAL,
  name VARCHAR(100),
  phone TEXT[]);
```

```
INSERT INTO table_array (name, phone)
VALUES ('Adam', ARRAY [ '(801)-123-4567', '(819)-555-2222' ]);

INSERT INTO table_array (name, phone)
VALUES ('Linda', ARRAY [ '(+34) 94 123 456', '(+34) 654 123 456' ]);
```

Use array keyword when inserting.



Note the data type has square brackets after indicating it is an array.

```
SELECT * FROM table_array;
```

	id integer	name character varying (100)	phone text[]
1	1	Adam	{(801)-123-4567,(819)-555-2222}
2	2	Linda	{'(+34) 94 123 456','(+34) 654 123 456'}

Note that the array is stored inside curly brackets like an object.

```
{
(801)-123-4567,
(819)-555-2222
}
```

```
SELECT name, phone[1] FROM table_array;
```

I can select the first element of the array by specifying its position in the array.

	name character varying (100)	phone text
1	Adam	(801)-123-4567
2	Linda	(+34) 94 123 456

```
SELECT * FROM table_array WHERE phone[2] = '(819)-555-2222';
```

	id integer	name character varying (100)	phone text[]
1	1	Adam	{(801)-123-4567,(819)-555-2222}

I can use an array position in a where clause to find a row. The query will look at all of the second elements in the array to find the matching row.

5.7 hstore

Hstore is a data type that stores the data into key-value pairs. The hstore module implements the hstore datatype. The keys and values are just text strings only.

`CREATE EXTENSION IF NOT EXISTS hstore;` Install the hstore extension on the database.

```
CREATE TABLE table_hstore (  
  book_id SERIAL PRIMARY KEY,  
  title VARCHAR(100) NOT NULL,  
  book_info HSTORE);
```

book_info

General Definition Constraints Variables Security SQL

Data type

Length/Precision

```
INSERT INTO table_hstore (title, book_info) VALUES  
(  
  'title 1',  
  "publisher" => "ABC publisher",  
  "paper_cost" => "10.00",  
  "e_cost" => "5.85"  
);
```

```
SELECT * FROM  
table_hstore;
```

	book_id [PK] integer	title character varying (100)	book_info hstore
1	1	title 1	"e_cost"=>"5.85", "publisher"=>"ABC publisher", "paper_cost"=>"10.00"

```
INSERT INTO table_hstore (title, book_info) VALUES  
(  
  'title 2',  
  "publisher" => "XYZ publisher",  
  "paper_cost" => "20.00",  
  "e_cost" => "11.85"  
);
```



```
SELECT book_info -> 'publisher' AS publisher FROM table_hstore;
```

I can query a particular value from a key-value pair within the object by using -> to name the key within single quotes.

	publisher text	
1	ABC publisher	
2	XYZ publisher	

```
SELECT  
book_info -> 'publisher' AS publisher,  
book_info -> 'e_cost' AS electronic_cost  
FROM table_hstore;
```

	publisher text	electronic_cost text
1	ABC publish...	5.85
2	XYZ publisher	11.85

5.7 JSON

PostgreSQL has built-in support for JSON with great range of processing functions and operators, and complete indexing support.

The JSON Data type is actually text under the hood, with a verification that the format is valid JSON input much like XML

The JSONB implemented a binary version of the JSON datatype

The JSON data type, being a data type, stores the data presentation exactly as it is sent to PostgreSQL, including whitespace and indentation, and also multiple keys when present (no processing at all is done on the content, only form validation)

The JSON data type is an advanced binary storage format with full processing, indexing and searching capabilities, and as such pre-processes the JSON data to an internal format, which does include a single value per key; and also isn't sensible to extra whitespace or indentation.

```
CREATE TABLE table_json (  
  id SERIAL PRIMARY KEY,  
  docs JSON);
```

```
INSERT INTO table_json (docs) VALUES  
( '[1,2,3,4,5,6]' ),  
( '[2,3,4,5,6,7]' ),  
( '{ "key": "value" }' );
```

```
SELECT * FROM table_json;
```

	id [PK] integer	docs json
1	1	[1,2,3,4,5,6]
2	2	[2,3,4,5,6,7]
3	3	{ "key": "value" }

```
SELECT * FROM table_json WHERE docs @> '2';  
ERROR: operator does not exist: json @> unknown  
LINE 1: SELECT * FROM table_json WHERE docs @> '2';  
                                              ^
```

HINT: No operator matches the given name and argument types. You might need to add explicit type casts.

The reason for this failure is that it is stored as JSON and not JSONB

```
ALTER TABLE table_json  
ALTER COLUMN docs TYPE JSONB;
```

```
SELECT * FROM table_json WHERE docs @> '2';
```

Note how the @operator can search inside a JSONB object

	id [PK] integer	docs jsonb
1	1	[1, 2, 3, 4, 5, 6]
2	2	[2, 3, 4, 5, 6, 7]

```
CREATE INDEX ON table_json USING GIN (docs jsonb_path_ops);
```

By default, PostgreSQL creates a btree index but we can specify a GIN index and then create an index on the JSONB column. This will speed up searching for values within the JSON objects using @.

```
SELECT * FROM table_json WHERE docs @> '2';
```

This is it is going to use the index to search, not the normal selecting process.

5.8 Network Addresses

PostgreSQL offers Data types to store IPv4, IPv6 and MAC addresses.

It is better to use these types instead of TEXT to store network addresses because these types offer input error checking and specialised operators and functions.

Name	Storage Size	Notes
cidr	7 or 19 bytes	IPv4 & IPv6 networks
inet	7 or 19 bytes	IPv4 & IPv6 hosts & networks
macaddr	6 bytes	Mac Addresses
macaddr8	8 bytes	Mac Addresses (EUI-64 format)

When sorting cidr or inet, IPv4 addresses will always sort before IPv6 addresses, including IPv4 addresses encapsulated or mapped to IPv6 addresses.

These types are bundled with indexing support and advanced functions and operator support.

```
CREATE TABLE
table_netaddr (
id SERIAL PRIMARY KEY,
ip INET);
```

```
INSERT INTO table_netaddr (ip)
VALUES
('4.35.221.243'),
('4.152.207.126'),
('4.152.207.238'),
('12.1.223.132'),
('12.8.192.60');
```

```
SELECT * FROM table_netaddr;
```

	id [PK] integer	ip inet
1	1	4.35.221.243
2	2	4.152.207.126
3	3	4.152.207.238
4	4	12.1.223.132
5	5	12.8.192.60

	ip inet	inet_24 inet
1	4.35.221.243	4.35.221.243/24
2	4.152.207.126	4.152.207.126/24
3	4.152.207.238	4.152.207.238/24
4	12.1.223.132	12.1.223.132/24
5	12.8.192.60	12.8.192.60/24

```
SELECT
ip,
set_masklen(ip,24) AS inet_24
FROM table_netaddr;
```

Set_masklen can be used to output an IP with the CIDR/mask notation

```
SELECT
ip,
set_masklen(ip,24) AS inet_24,
set_masklen(ip::cidr,24) as cidr_24,
set_masklen(ip::cidr,28) as cidr_28
FROM table_netaddr;
```

I can cast an IP address to get the network address and subnet mask for a specific IP address.

	ip inet	inet_24 inet	cidr_24 cidr	cidr_28 cidr
1	4.35.221.243	4.35.221.243/24	4.35.221.0/24	4.35.221.240/28
2	4.152.207.126	4.152.207.126/24	4.152.207.0/24	4.152.207.112/28
3	4.152.207.238	4.152.207.238/24	4.152.207.0/24	4.152.207.224/28
4	12.1.223.132	12.1.223.132/24	12.1.223.0/24	12.1.223.128/28
5	12.8.192.60	12.8.192.60/24	12.8.192.0/24	12.8.192.48/28

6. Modify Tables, Constraints & Type Conversion

[6.1 Add/modify Table Columns](#)

[6.2 Add Constraint to Column](#)

[6.3 Data Type Conversion](#)

[6.4 Using CAST for Data Conversion](#)

[6.5 Implicit to Explicit Data Conversions](#)

[6.6 Table Data Conversion](#)

[6.7 to char Conversion Function](#)

[6.8 to number Conversion Function](#)

[6.9 to date Conversion Function](#)

[6.10 to timestamp Conversion Function](#)

6.1 Add/modify Table Columns

```
CREATE DATABASE mydata
WITH
OWNER = postgres
ENCODING = 'UTF8'
LC_COLLATE = 'C'
LC_CTYPE = 'C'
TABLESPACE = pg_default
CONNECTION LIMIT = -1;
```

```
CREATE TABLE persons (
person_id SERIAL PRIMARY KEY,
first_name VARCHAR(20) NOT NULL,
last_name VARCHAR(20) NOT NULL
);
```

```
ALTER TABLE persons
ADD COLUMN age int NOT NULL,
ADD COLUMN nationality VARCHAR(20)
NOT NULL,
ADD COLUMN email VARCHAR(100)
UNIQUE;
```

```
ALTER TABLE persons
RENAME TO users;
```

```
ALTER TABLE userspersons
RENAME TO persons;
```

```
ALTER TABLE persons
RENAME COLUMN age TO
person_age;
```

```
ALTER TABLE persons
DROP COLUMN person_age;
```

```
ALTER TABLE persons
ADD COLUMN age
VARCHAR(10);
```

```
ALTER TABLE persons
ALTER COLUMN age TYPE INT;
ERROR: column "age" cannot be
cast automatically to type
integer
HINT: You might need to
specify "USING age::integer".
```

```
ALTER TABLE persons
ALTER COLUMN age TYPE INT
USING age::int;
```

```
ALTER TABLE persons
ALTER COLUMN age TYPE
VARCHAR(20);
```

```
ALTER TABLE persons
ALTER COLUMN age TYPE INT
USING age::int;
```

```
ALTER TABLE persons
ADD COLUMN is_enable VARCHAR(1);
```

```
ALTER TABLE persons
ALTER COLUMN is_enable SET
DEFAULT 'Y';
```

```
INSERT INTO persons
(first_name,
Last_name,
nationality, age)
VALUES
('John',
'Smith',
'US',
'40');
```

	person_id [PK] integer	first_name character varying (20)	last_name character varying (20)	nationality character varying (20)	email character varying (100)	age integer	is_enable character varying (1)
1	1	John	Smith	US	[null]	40	Y

6.2 Add Constraint to Column

```
CREATE TABLE web_links (  
  link_id SERIAL PRIMARY KEY,  
  link_url VARCHAR(255) NOT NULL,  
  link_target VARCHAR(20)  
);
```

```
INSERT INTO web_links  
(link_url, link_target)  
VALUES ('https://www.google.com', '_blank');
```

```
ALTER TABLE web_links  
ADD CONSTRAINT unique_web_url UNIQUE (link_url);
```

```
INSERT INTO web_links  
(link_url, link_target)  
VALUES ('https://www.google.com', '_blank');  
ERROR: Key (link_url)=(https://www.google.com) already exists.duplicate key value  
violates unique constraint "unique_web_url"  
ERROR: duplicate key value violates unique constraint "unique_web_url"
```

```
INSERT INTO web_links  
(link_url, link_target)  
VALUES ('https://www.amazon.com', '_blank');
```

```
INSERT INTO web_links  
(link_url, link_target, is_enabled)  
VALUES ('https://www.citi.com', '_blank', 'Q');  
ERROR: Failing row contains (5, https://www.citi.com, _blank, Q).new row for relation  
"web_links" violates check constraint "web_links_is_enabled_check" ERROR: new row for  
relation "web_links" violates check constraint "web_links_is_enabled_check"
```

```
INSERT INTO web_links  
(link_url, link_target, is_enabled)  
VALUES ('https://www.citi.com', '_blank', 'Y');
```



```
ALTER TABLE web_links
ADD COLUMN is_enabled VARCHAR(2);

INSERT INTO web_links
(link_url, link_target, is_enabled)
VALUES ('https://www.netflix.com', '_blank', 'Y');

ALTER TABLE web_links
ADD CHECK (is_enabled IN ('Y', 'N'));
```

```
UPDATE web_links
SET is_enabled = 'Y'
WHERE link_id = 1;
```

```
UPDATE web_links
SET is_enabled = 'Q'
WHERE link_id = 3;
```

```
ERROR: Failing row contains (3, https://www.amazon.com, _blank, Q).new row for relation
"web_links" violates check constraint "web_links_is_enabled_check"
ERROR: new row for relation "web_links" violates check constraint
"web_links_is_enabled_check"
```

```
UPDATE web_links
SET is_enabled = 'Y'
WHERE link_id = 3;
```

6.3 Data Type Conversion

A data conversion is when data is converted from its ORIGINAL data type to ANOTHER data type. There are two types of Conversion:

IMPLICIT - Data conversion is done automatically

EXPLICIT - Data conversion is done via 'conversion functions' i.e. CAST or ::

```
SELECT * FROM movies WHERE movie_id =1;  
SELECT * FROM movies WHERE movie_id = '1';
```

	movie_id [PK] integer	movie_name character varying (100)	movie_length integer	movie_lang character varying (20)	age_certificate character varying (10)	release_date date	director_id integer
1	1	A Clockwork Orange	112	English	18	1972-02-02	13

Note that the movie_id column is an integer, yet when I try and select a column with value of movie_id equal to a string it still runs. This is an example of IMPLICIT data type conversion because postgreSQL is doing the type conversion automatically.

```
SELECT * FROM movies WHERE movie_id = INTEGER '1';
```

The same query but now I am EXPLICITLY saying that the value is an INTEGER so that postgreSQL EXPLICITLY performs the data conversion of string to integer.

6.4 Using CAST for Data Conversion

PostgreSQL CAST operator is used to convert a value of one type to another data type.

CAST (expression AS target_data_type);

Expression - can be a constant, a table column, or an expression

target_data_type - Boolean, Character (char, varchar, text), Numeric (integer, floating point number), array, JSON, UUID, hstore

```
SELECT CAST('10' AS INTEGER);
```

	int4 integer
1	10

```
SELECT CAST('10n' AS INTEGER);  
ERROR: invalid input syntax for type  
integer: "10n"  
LINE 1: SELECT CAST('10n' AS INTEGER);
```

The expression does not conform the the receiving type

```
SELECT  
CAST ('2020-01-01' AS DATE),  
CAST ('01-MAY-2022' AS DATE);
```

	date date	date date
1	2020-01-01	2022-05-01

String to date conversion

```
SELECT  
CAST ('true' AS BOOLEAN),  
CAST ('false' AS BOOLEAN),  
CAST ('T' AS BOOLEAN),  
CAST ('F' AS BOOLEAN),  
CAST ('1' AS BOOLEAN),  
CAST ('0' AS BOOLEAN),  
CAST ('Y' AS BOOLEAN),  
CAST ('N' AS BOOLEAN);
```

	bool boolean	bool boolean	bool boolean	bool boolean	bool boolean	bool boolean	bool boolean	bool boolean
1	true	false	true	false	true	false	true	false

String to Boolean conversion

```
SELECT
CAST ('14.7888' AS DOUBLE PRECISION);
```

	float8 double precision
1	14.7888

String to double precision

```
SELECT
'10'::INTEGER,
'2020-01-01'::DATE,
'01-MAY-2022'::DATE;
```

	int4 integer	date date	date date
1	10	2020-01-01	2022-05-01

I can also use the following expression::type

```
SELECT
'2020-01-01 18:30:25.227'::TIMESTAMP;
```

	timestamp timestamp without time zone
1	2020-01-01 18:30:25.227

String to timestamp

```
SELECT
'2020-01-01 18:30:25.227'::TIMESTAMPTZ;
```

	timestamptz timestamp with time zone
1	2020-01-01 18:30:25.227+01

String to timestamp with timezone. Note how it has used my local time because no TZ is specified.

```
SELECT
'10 minute'::INTERVAL,
'4 hour'::INTERVAL,
'1 day'::INTERVAL,
'2 week'::INTERVAL,
'5 month'::INTERVAL;
```

	interval interval	interval interval	interval interval	interval interval	interval interval
1	00:10:00	04:00:00	1 day	14 days	5 mons

String to Interval

6.5 Implicit to Explicit Data Conversions

```
SELECT  
factorial(20) AS factorial,  
CAST(factorial(20) AS BIGINT);
```

	factorial numeric	factorial bigint
1	2432902008176640000	2432902008176640000

Factorial to bigint

```
SELECT  
ROUND(10.4) AS "result",  
ROUND (CAST(10 AS NUMERIC),4) AS "result";
```

	result numeric	result numeric
1	10	10.0000

Round with Numeric

```
SELECT  
ROUND(10.4) AS "result",  
ROUND (CAST(10 AS NUMERIC),4) AS "result";
```

	result numeric	result numeric
1	10	10.0000

Round with Numeric

```
SELECT  
SUBSTR('123456',2) AS "implicit",  
SUBSTR(CAST('123456' AS TEXT),2) AS "explicit";
```

	implicit text	explicit text
1	23456	23456

Take a section of a string and cast it as text

6.6 Table Data Conversion

```
CREATE TABLE ratings (  
  rating_id SERIAL PRIMARY KEY,  
  rating VARCHAR(1) NOT NULL  
);
```

```
INSERT INTO ratings (rating)  
VALUES ('A'),('B'),('C'),('D');  
  
INSERT INTO ratings (rating)  
VALUES (1),(2),(3),(4);
```

The scenario is that the company wants all ratings to now be of a numeric type so that when a query is performed if the return is a letter (A,B,C or D) then it is converted into a number (1,2,3, or 4).

	rating_id [PK] integer	rating character varying (1)
1	1	A
2	2	B
3	3	C
4	4	D
5	5	1
6	6	2
7	7	3
8	8	4

```
SELECT  
  rating_id,  
  CASE  
    WHEN rating~E'^\\d+$' THEN  
      CAST (rating AS INTEGER)  
    ELSE  
      0  
    END AS rating  
FROM  
ratings;
```

	rating_id [PK] integer	rating integer
1	1	0
2	2	0
3	3	0
4	4	0
5	5	1
6	6	2
7	7	3
8	8	4

The case statement is used to cast any numeric value as an integer and anything else as a zero.

6.7 to_char Conversion Function

Valid options for Numbers are described by a number, symbol or letter code:

9	Numeric value with the specified number of digits
0	Numeric value with leading zero's
.	(period) decimal point that uses locale
D	Decimal point that uses locale
,	(coma) group thousand separator
FM	Fill mode, which supresses padding blanks and leading zero's
PR	Negative values in angle brackets
S	Sigh anchored to a number that uses locale
L	Currency symbol that uses locale
G	Group separator that uses locale
MI	Minus sign in the specified position for numbers that are less than zero
PL	Plus sign in the specified position for numbers that are greater than 0
SG	Plus / Minis sign in the specified position
RN	Roman numeral that ranges from 1 to 3999
TH or th	Upper case or lower case ordinal number suffix

PostgreSQL TO_CHAR() function converts a timestamp, an interval, an integer, a double precision or a numeric value to a string.

TO_CHAR(expression, format)

Return value will be a text type

Valid timestamp format options:	
Y,YYY	Year in 4 digits with a coma
YYYY	Year in 4 digits
YYY	last three digits of year
YY	Last Two digits of year
IYYY	ISO 8601 week-numbering year (4 or more digits)
IYY	Last three digits of ISO 8601 week-numbering year
IY	Last two digits of ISO 8601 week-numbering year
I	Last Digit of ISO 8601 week-numbering year
B.C., b.c., A.D., or a.d.	Era indicators without periods
MONTH	English month name in uppercase
Month	Full capitalised month English month name
month	month Full lowercase English month name
MON	Abbreviated uppercase month name (JAN FEB MAR...)
Mon	Abbreviated Capitalised lowercase month name (Jan Feb Mar...)
mon	Abbreviated lowercase month name (jan feb mar...)
MM	Month Number from 01 to 12
DAY	Full uppercase day name
Day	Full capitalises day name

Valid timestamp format options (continued):	
day	Full lowercase day name
DY	Abbreviated uppercase day name
Dy	Abbreviate capitalised day name
DDD	Day of year (001 - 366)
IDDD	IDDD Day of Year ISO 8601 week-numbering year (001-371): Day 1 of the year is the Monday of the first week of the year
DD	Day of Month (0 - 31)
D	Day of week: Sunday (1) to Saturday (7)
ID	ID ISO 8601 day of the week: Sunday (1) to Saturday (7)
W	Week of month (1-5) The first week starts on the first day of the month
WW	Week number of year (1-53) The first week starts on the first day of year
IW	IW Week number of ISO 8601 week-numbering year (01-53): The first Thursday of the year is in week one
CC	Century eg 20, 21, 22 etc
J	Julian day (integer days since November 24th 4714 BC at Midnight UTC)
RM	Month in uppercase Roman Numerals (I -XII)
rm	Month in Lower case roman numerals (i - xii)
HH	HH Hour of day (0-12)
HH12	Hour of Day (0-12)
HH24	Hour of day (0-23)


```
SELECT
TO_CHAR (100870,'9,99999') AS "5_digits",
TO_CHAR (100870,'9,9999') AS "4_digits";
```

	5_digits text	4_digits text
1	1,00870	#,####

Note that I am using the 9 and the Coma selectors to convert this number into a text string.

Note how if I only supply 4 9's after the coma then this is not enough to convert this number to text and it will signify this by marking it as hashes.

```
SELECT release_date,
TO_CHAR(release_date, 'DD-MM-YY'),
TO_CHAR(release_date, 'Dy, MM, YY')
FROM movies;
```

	release_date date	to_char text	to_char text
1	1972-02-02	02-02-72	Wed, 02, 72
2	1979-08-15	15-08-79	Wed, 08, 79

Note how I can use date formats to significantly change how a date output appears

```
SELECT TO_CHAR (
TIMESTAMP '2020-01-01 10:30:59',
'HH24:MI:SS');
```

	to_char text
1	10:30:59

I can use time formats to change a full timestamp.

```
SELECT movie_id, revenues_domestic,
TO_CHAR(revenues_domestic, '$99999D99')
FROM movies_revenues;
```

	movie_id integer	revenues_domestic numeric (10,2)	to_char text
1	45	22.20	\$ 22.20
2	13	199.40	\$ 199.40

For Null values it will still return null

I am prefixing the output with the dolar sign then converting to 5 numbers then decimal (D) then two numbers.

6.8 to_number Conversion Function

```
SELECT  
TO_NUMBER('1420.99', '9999.99') AS "two_decimal_places",  
TO_NUMBER('1420.99', '9999.9') AS "one_decimal_place",  
TO_NUMBER('1420.99', '9999') AS "no_decimal_places";
```

	two_decimal_places numeric	one_decimal_place numeric	no_decimal_places numeric
1	1420.99	1420.9	1420

PostgreSQL TO_NUMBER() function converts a character string into a numeric value. TO_NUMBER(string, format) where the format options are the same as those on [slide 74](#)

```
SELECT  
TO_NUMBER('10,625.78-', '99G999D99S');
```

	to_number numeric
1	-10625.78

9	Numeric value with the specified number of digits
D	Decimal point that uses locale
G	Group separator that uses locale
S	Sign anchored to a number that uses locale

I am using 9 to represent numbers and the number of digits, G for a group separator, D for decimal and S to change the sign.

```
SELECT  
TO_NUMBER('$10,625.78', 'L99G999D99');
```

	to_number numeric
1	10625.78

Here I am using a preceding L to represent the dolar sign in the text.

```
SELECT  
TO_NUMBER('$1,420.64', 'L9G999D99'),  
TO_NUMBER('1,234,567.89', '9G999G999D99');
```

	to_number numeric	to_number numeric
1	1420.64	1234567.89

6.9 to_date Conversion Function

```
SELECT  
TO_DATE('2020/10/22', 'YYYY/MM/DD'),  
TO_DATE('022199', 'MMDDYY'),  
TO_DATE('March 07. 2019', 'Month DD, YYYY');
```

	to_date date	to_date date	to_date date
1	2020-10-22	1999-02-21	2019-03-07

PostgreSQL TO_DATE() function converts a character string into a date value.

TO_DATE(string, format) where the format options are the same as those on [slide 75](#)

Note that the date format in PostgreSQL is always in default of YYYY-MM-DD

```
SELECT  
TO_DATE('2020/02/30', 'YYYY/MM/DD');  
ERROR: date/time field value out  
of range: "2020/02/30"
```

In this example it is throwing an error because February can never have 30 days

```
SELECT  
TO_DATE('30th November 1979', 'DDth Month YYYY'),  
TO_DATE('1st April 2020', 'DDth Month YYYY');
```

Note that if the string has spaces used them also in the date format. Th or TH is used to represent dayth or dayst formats.

	to_date date	to_date date
1	1979-11-30	2020-04-01

6.10 to_timestamp Conversion Function

```
SELECT
TO_TIMESTAMP ('2020-10-28 10:30:25', 'YYYY-MM-DD HH:MI:SS'),
TO_TIMESTAMP ('2020      MAY', 'YYYY Mon');
```

	to_timestamp timestamp with time zone 	to_timestamp timestamp with time zone 
1	2020-10-28 10:30:25+01	2020-05-01 00:00:00+02

Note how in the string, even if the timestamp has spaces, or lots of spaces, postgreSQL ignores the whitespace and only focuses on converting the text. If there is missing information such as the day and time then it will pad it out with default values.

```
SELECT
TO_TIMESTAMP ('2020-01-01 32:8:00', 'YYYY-MM-DD HH24:MI:SS');
ERROR: date/time field value out of range: "2020-01-01 32:8:00"
```

PostgreSQL TO_TIMESTAMP() function to convert a string to a timestamp based on specific formats found on [slide 75](#).

TO_TIMESTAMP(string, format)

Minimal error checking is provided so ensure that date and time components are actually valid.

7. User Defined Data Types

[7.1 Create Domain Data Type for an Address](#)

[7.2 Create Domain Data Type for a positive Number](#)

[7.3 Create Domain Data Type for a Post Code](#)

[7.4 Create Domain Data Type for Email Validation](#)

[7.5 Create Domain Data Type for Enum or Set of Values](#)

[7.6 Get List of All Domain Data Types](#)

[7.7 Drop a Domain Data Type](#)

[7.8 Create a Composite Address Object](#)

[7.9 Create a Composite Inventory item Data Type](#)

[7.10 Create an Enum Data Type and See how to Drop](#)

[7.11 Alter a Composite Data Type, Change Schema & More](#)

[7.12 Alter Type, Alter an Enum Data Type](#)

[7.13 Alter Enum Data in a Production Server](#)

[7.14 An Enum with a Default Value in a Table](#)

[7.15 Create a Type If Not Exists Using PL/pgSQL Function](#)

7.1 Create Domain Data Type for an Address

CREATE DOMAIN statement creates a user-defined data type with a range, optional DEFAULT, NOT NULL and CHECK constraint.

They have to be unique within a schema scope, cannot be re-used outside of scope where they are defined.

Help to STANDARDISE your database data types in one place.

A domain Data Type is a COMMON data type and can be RE-USED in multiple columns. Write once and share

Null is default

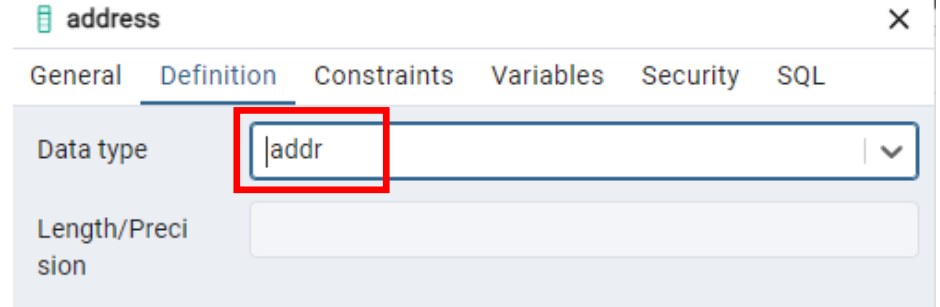
Composite type: Only single value return.

Syntax of: **CREATE DOMAIN name AS data_type constraint**

```
CREATE DOMAIN addr VARCHAR(100) NOT NULL
```

```
CREATE TABLE locations (  
  address addr,  
);
```

It is not a standard data type but the one we defined.



address

General Definition Constraints Variables Security SQL

Data type |addr|

Length/Precision

```
INSERT INTO locations (address) VALUES ('123 London');  
SELECT * FROM locations;
```

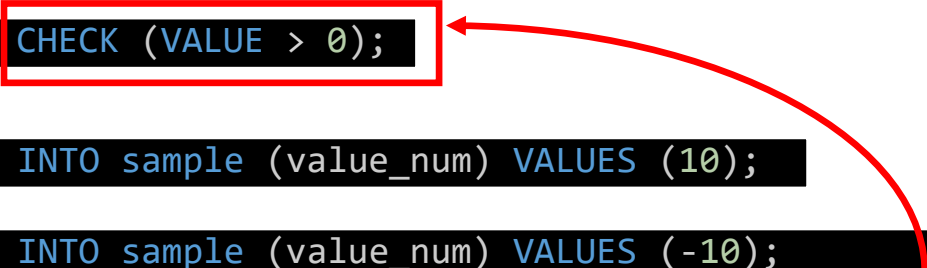
	address character varying (100)
1	123 London

```
INSERT INTO locations (address) VALUES ('123 London 123 London 123 London 123 London 123  
London 123 London 123 London 123 London 123 London 123 London 123 London');  
ERROR: value too long for type character varying(100)
```

Although the address column is our own defined data type called addr, we defined addr to have a type of varchar(100). It has inherited the properties of the domain data type.

7.2 Create Domain Data Type for a positive Number

```
CREATE DOMAIN positive_numeric INT NOT NULL CHECK (VALUE > 0);
```



```
CREATE table sample (  
  Sample_id SERIAL PRIMARY KEY,  
  value_num positive_numeric  
);
```

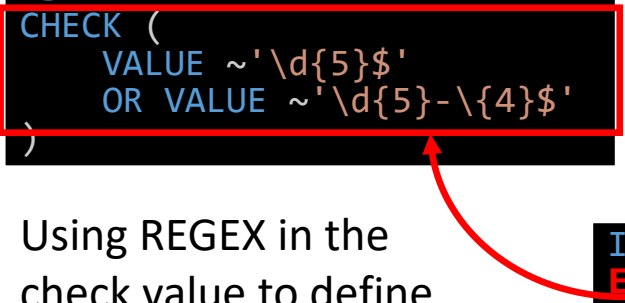
```
INSERT INTO sample (value_num) VALUES (10);
```

```
INSERT INTO sample (value_num) VALUES (-10);  
ERROR: value for domain positive_numeric violates  
check constraint "positive_numeric_check"
```

The advantage of using a DOMAIN data type is that we can re use them multiple times in multiple columns. For example whenever we need a positive number data type.

7.3 Create Domain Data Type for a Post Code

```
CREATE DOMAIN us_postal_code  
AS TEXT  
CHECK (  
  VALUE ~ '\d{5}$'  
  OR VALUE ~ '\d{5}-\d{4}$'  
);
```



```
CREATE TABLE addresses (  
  address_id SERIAL PRIMARY KEY,  
  postal_code us_postal_code  
);
```

```
INSERT INTO addresses  
(postal_code) VALUES ('10000');
```

Using REGEX in the
check value to define
acceptable values

```
INSERT INTO addresses (postal_code) VALUES ('10000-1000-1000');  
ERROR: value for domain us_postal_code violates check  
constraint "us_postal_code_check"
```


7.4 Create Domain Data Type for Email Validation

```
CREATE DOMAIN proper_email VARCHAR(150)
CHECK (VALUE ~* '^[A-Za-z0-9._%~]+@[A-Za-z0-9.-]+[.]+[A-Za-z]+$');
```

Using REGEX in the check value to define acceptable values

```
CREATE TABLE client_names (
  client_name_id SERIAL PRIMARY KEY,
  email proper_email
);
```

```
INSERT INTO client_names (email) VALUES ('a@b.com');
```

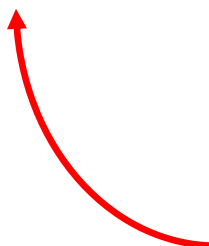
```
INSERT INTO client_names (email) VALUES ('a @b.com');
ERROR: value for domain proper_email violates
check constraint "proper_email_check"
```

7.5 Create Domain Data Type for Enum or Set of Values

```
CREATE DOMAIN valid_colour VARCHAR(10)
CHECK (VALUE IN ('red', 'green', 'blue'));
```

```
CREATE TABLE colours (
  colour valid_colour);
```

Using IN to define a list
of acceptable values



```
INSERT INTO colours (colour) VALUES ('red');
```

```
INSERT INTO colours (colour) VALUES ('yellow');
ERROR: value for domain valid_colour
violates check constraint
"valid_colour_check"
```

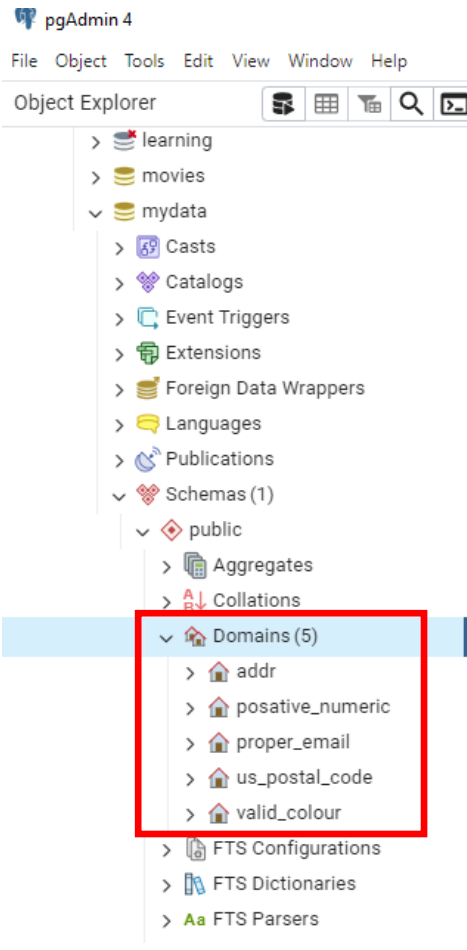
```
CREATE DOMAIN user_status VARCHAR(10)
CHECK (VALUE IN ('enable', 'dissable', 'temp'));
```

```
CREATE TABLE users (
  user_id SERIAL PRIMARY KEY
  user_status user_status);
```

7.6 Get List of All Domain Data Types

In PgAdmin 4 go to the database → Schema → Domains- upon expanding domains there is a list of all the manually created Domain Data Types.

Right Clicking on the domain → Properties will allow me to adjust the domain.



The screenshot shows the pgAdmin 4 interface. In the Object Explorer on the left, the 'public' schema is expanded, and the 'Domains (5)' folder is highlighted with a red box. The domains listed are: addr, positive_numeric, proper_email, us_postal_code, and valid_colour.

```
SELECT typename
FROM pg_catalog.pg_type
JOIN pg_catalog.pg_namespace
ON pg_namespace.oid = pg_type.typnamespace
WHERE typtype = 'd' and nspname = 'public';
```

The above query will find the names of all domain 'd' data types

	typename
1	addr
2	positive_numeric
3	us_postal_code
4	proper_email
5	valid_colour

7.7 Drop a Domain Data Type

```
DROP DOMAIN positive_numeric;  
ERROR: column value_num of table sample depends on type positive_numeric  
cannot drop type positive_numeric because other objects depend on it  
  
ERROR: cannot drop type positive_numeric because other objects depend on it SQL  
state: 2BP01 Detail: column value_num of table sample depends on type  
positive_numeric Hint: Use DROP ... CASCADE to drop the dependent objects too.
```

Be cautious about dropping Domains because they could be used in multiple tables. if I try and drop an in use domain I will get an error message that specifies where it is being used, i.e. **column value_num of table sample**.

```
SELECT * FROM sample;
```

	sample_id [PK] integer	value_num integer
1	1	10

```
DROP DOMAIN positive_numeric CASCADE;  
NOTICE: drop cascades to column value_num of table sample DROP DOMAIN
```

If I run the drop Domain with CASCADE option as suggested then it will delete the domain and **ALSO DELETE THE TABLE COLUMN value_num of table sample**.

```
SELECT * FROM sample;
```

	sample_id [PK] integer
1	1

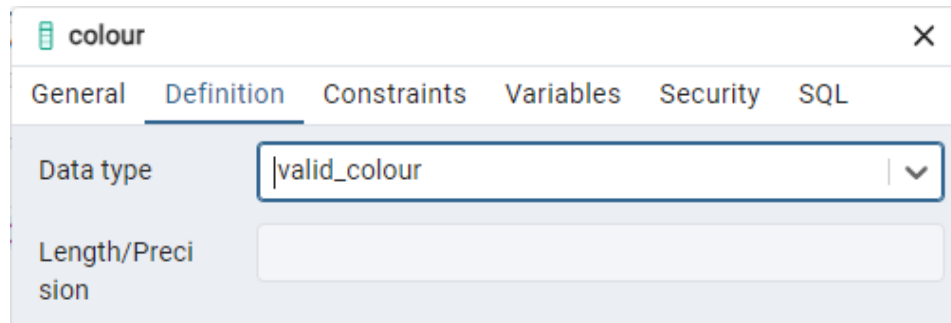
```
DROP DOMAIN valid_colour;
```

ERROR: column colour of table colours depends on type valid_colour

cannot drop type valid_colour because other objects depend on it

ERROR: cannot drop type valid_colour because other objects depend on it

SQL state: 2BP01 Detail: column colour of table colours depends on type valid_colour
Hint: Use DROP ... CASCADE to drop the dependent objects too.



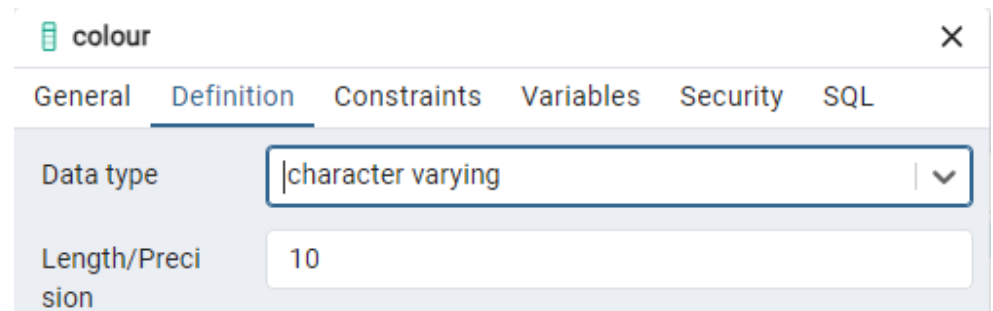
colour

General Definition Constraints Variables Security SQL

Data type |valid_colour|

Length/Precision

```
ALTER TABLE colour  
ALTER COLUMN colour TYPE VARCHAR(10);
```



colour

General Definition Constraints Variables Security SQL

Data type |character varying|

Length/Precision 10

To safely remove a Domain first change the data type of any columns that are using it.

```
DROP DOMAIN Valid_colour;
```

Now I can drop the Domain because no columns are referencing it.
I do not need the cascade option.

```
SELECT * FROM colours;
```

	colour
	character varying (10)
1	red

Without losing Data. **NEVER USE CASCADE IN A PRODUCTION DATABASE.**

7.8 Create a Composite Address Object

Composite data types is a list of fields with corresponding data types

It is used in a table as a 'column'

Used in functions or procedures

Can return multiple values, it is a composite data type

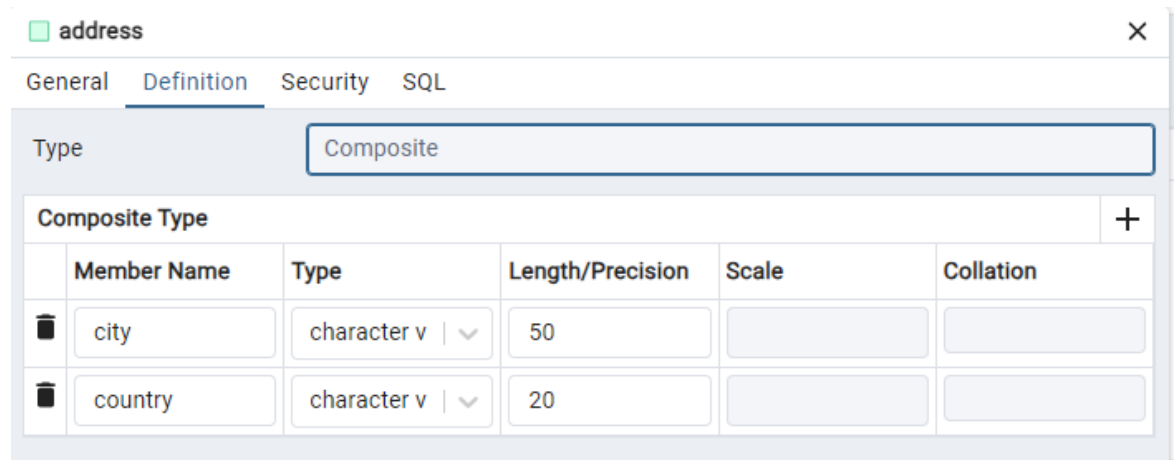
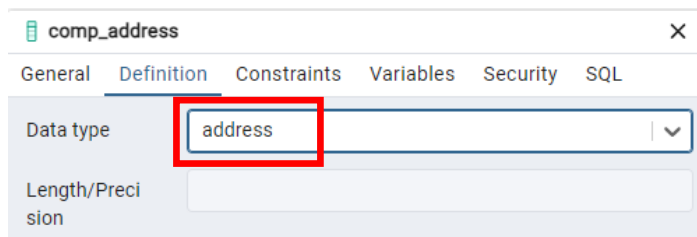
Composite Data type Syntax: **CREATE TYPE name AS (field_columns_properties);**

```
CREATE TYPE address AS (  
  city VARCHAR(50),  
  country VARCHAR(20));
```

To view the composite data type in PgAdmin 4 go to the database
→ Types. Right clicking on it then → properties

```
CREATE TABLE companies (  
  comp_id SERIAL PRIMARY KEY,  
  comp_address address  
);
```

I can use a composite data type as a column in a table.



I can see that it is using the composite data type that I created.

```
INSERT INTO companies (comp_address) VALUES (ROW('LONDON','UK'));
```

```
SELECT * FROM companies;
```

	comp_id [PK] integer	comp_address address
1	1	(LONDON,UK)

Notice how I use the ROW() function is used to insert multiple values into a composite column.

```
INSERT INTO companies (comp_address) VALUES (ROW('NEW YORK', 'US'));
```

```
SELECT (comp_address).city, (comp_address).country FROM companies;
```

	city character varying (50)	country character varying (20)
1	LONDON	UK
2	NEW YORK	US

I can select individual columns of the composite column by putting the composite column name in brackets then a dot followed by the sub column.

```
SELECT (companies.comp_address).city, (companies.comp_address).country FROM companies;
```

	city character varying (50)	country character varying (20)
1	LONDON	UK
2	NEW YORK	US

If I am using composite columns with a JOIN then I can also specify the table.

7.9 Create a Composite Inventory_item Data Type

```
CREATE TYPE inventory_item AS  
(  
  product_name VARCHAR(200),  
  supplier_id INT,  
  price NUMERIC  
);
```

```
CREATE table inventory (  
  inventory_id SERIAL PRIMARY KEY,  
  item inventory_item  
);
```

```
INSERT INTO inventory (item)  
VALUES (ROW('Pen','10',4.99));
```

```
INSERT INTO inventory (item)  
VALUES (ROW('Paper','20',10.99));
```

```
SELECT * FROM inventory;
```

	inventory_id [PK] integer	item inventory_item
1	1	(Pen,10,4.99)
2	2	(Paper,20,10.99)

```
SELECT (inventory.item).product_name  
FROM inventory  
WHERE (inventory.item).price < 3.99;
```

	product_name character varying (200)
1	Pen
2	Paper

```
SELECT (inventory.item).product_name  
FROM inventory  
WHERE (inventory.item).price < 5.99;
```

	product_name character varying (200)
1	Pen

I can define the composite column then the field within that composite column and use it in where clauses.

7.10 Create an Enum Data Type and See how to Drop

```
CREATE TYPE currency AS ENUM ('USD','EUR','GBP');
```

```
SELECT 'USD'::currency;
```

	currency
1	USD

```
ALTER TYPE currency ADD VALUE 'CHF' AFTER 'EUR';
```

I can add more values into the type. Note that the AFTER keyword will add the value after specified, not at the end which is the default.

```
CREATE TABLE stocks (  
  stock_id SERIAL PRIMARY KEY,  
  stock_currency currency  
);
```

```
INSERT INTO stocks (stock_currency) VALUES ('USD');
```

```
SELECT * FROM stocks;
```

	stock_id [PK] integer	stock_currency currency
1	1	USD

```
INSERT INTO stocks (stock_currency) VALUES ('USD1');  
ERROR: invalid input value for enum currency: "USD1"  
LINE 1: INSERT INTO stocks (stock_currency) VALUES ('USD1');
```

```
INSERT INTO stocks (stock_currency) VALUES ('CHF');
```

```
SELECT * FROM stocks;
```

	stock_id [PK] integer	stock_currency currency
1	1	USD
2	2	CHF

```
CREATE TYPE sample_type AS ENUM ('ABC','123');
```

```
DROP TYPE sample_type;
```

I can delete a type with the DROP command

7.11 Alter a Composite Data Type, Change Schema & More

Rename a data type	<code>ALTER TYPE name RENAME TO newname</code>
Change the owner	<code>ALTER TYPE name OWNER TO username</code>
Change the schema	<code>ALTER TYPE name SET SCHEMA schemaname</code>
To add a new attribute	<code>ALTER TYPE name ADD ATTRIBUTE attributes</code>

```
CREATE TYPE myaddress AS (  
  city VARCHAR(50),  
  country VARCHAR(20)  
);
```

I can create a new type and rename it.

```
ALTER TYPE myaddress RENAME TO my_address;
```

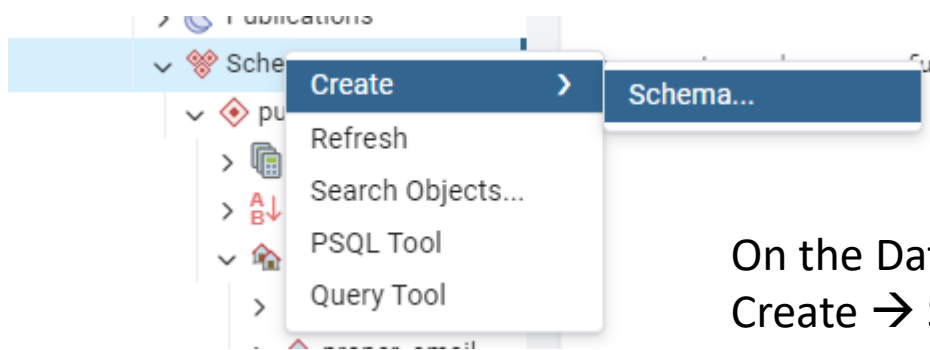
I can also change the owner of the type.

```
ALTER TYPE my_address OWNER TO Elliott;
```

The screenshot shows a window titled 'my_address' with tabs for 'General', 'Definition', 'Security', and 'SQL'. The 'General' tab is active, displaying the following fields:

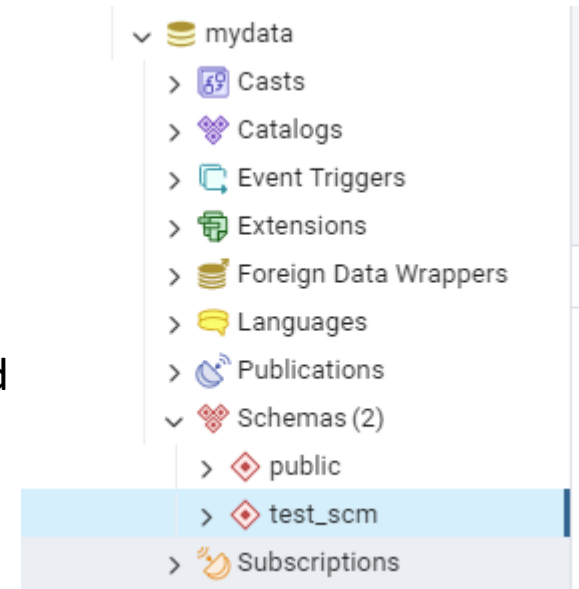
- Name: my_address
- Owner: **elliott** (highlighted with a red box)
- Schema: public
- Comment: (empty text area)

```
ALTER TYPE my_address OWNER TO postgres;
```



On the Database → Schemas →
Create → Schema.

I can create a new schema called
test_scm



```
ALTER TYPE my_address SET SCHEMA test_scm;
```

This will move the my_address TYPE from the
default PUBLIC schema to the test-scm schema.

I can add an attribute to the type

```
ALTER TYPE my_address ADD ATTRIBUTE street_address VARCHAR(150);  
ERROR: relation "my_address" does not exist
```

By default postgreSQL will assume the Type is in the public schema.

```
ALTER TYPE test_scm.my_address ADD ATTRIBUTE street_address VARCHAR(150);
```

Unless I specify the correct schema by name then dot then data type name

7.12 Alter Type, Alter an Enum Data Type

```
CREATE TYPE my_colours AS ENUM ('green','red','blue');
```

I can create a new Enum type and modify the values within.

```
ALTER TYPE my_colours RENAME VALUE 'red' TO 'orange';
```

I can show all the values of an Enum type. `SELECT enum_range(NULL::my_colours);`

	enum_range my_colours[]	
1	{green,orange,blue}	

```
ALTER TYPE my_colours ADD VALUE 'red' BEFORE 'green';
```

```
SELECT enum_range(NULL::my_colours);
```

	enum_range my_colours[]	
1	{red,green,orange,blue}	

```
ALTER TYPE my_colours ADD VALUE 'pink' AFTER 'green';
```

```
SELECT enum_range(NULL::my_colours);
```

	enum_range my_colours[]	
1	{red,green,pink,orange,blue}	

7.13 Alter Enum Data in a Production Server

```
CREATE TYPE status_enum AS ENUM ('queued','waiting','running','done');
```

```
SELECT * FROM jobs;
```

```
CREATE TABLE jobs (  
  job_id SERIAL PRIMARY KEY,  
  job_status status_enum  
);
```

```
INSERT INTO jobs (job_status) VALUES ('queued');  
INSERT INTO jobs (job_status) VALUES ('waiting');  
INSERT INTO jobs (job_status) VALUES ('running');  
INSERT INTO jobs (job_status) VALUES ('done');
```

I have a jobs table with a column called job_status which will take values from an Enum list

	job_id [PK] integer	job_status status_enum
1	1	queued
2	2	waiting
3	3	running
4	4	done

Lets say I want to remove the waiting status and change all rows in the jobs table that have a waiting status to running.

```
UPDATE jobs SET job_status = 'running' WHERE job_status = 'waiting';
```

Step 1: Set all the job_status columns to the new value where they match the old value.

```
ALTER TYPE status_enum RENAME TO status_enum_old;
```

Step 2: Conserve the old Enum values table by renaming it.

```
SELECT * FROM jobs;
```

	job_id [PK] integer	job_status status_enum
1	1	queued
2	3	running
3	4	done
4	2	running

```
CREATE TYPE status_enum_new AS ENUM ('queued','running','done');
```

Step 3: Create a new Enum type

```
ALTER TABLE jobs ALTER COLUMN job_status TYPE status_enum_new;
```

ERROR: column "job_status" cannot be cast automatically to type status_enum_new
HINT: You might need to specify "USING job_status::text::status_enum_new".

Step 4: assign the new type to the column

```
ALTER TABLE jobs  
ALTER COLUMN job_status  
TYPE status_enum_new  
USING job_status::text::status_enum_new;
```

Now the jobs table job status column is only accepting values from the satus_enum_new list.

```
DROP TYPE status_enum_old;
```

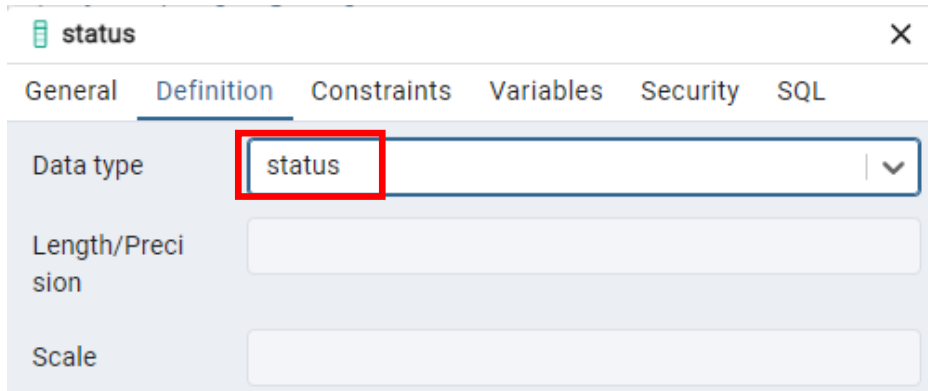
Step 5: Remove the old Enum list

7.14 An Enum with a default Value in a Table

```
CREATE TYPE status AS ENUM ('pending','approved','declined');
```

```
CREATE TABLE cron_jobs (  
  cron_job_id INT,  
  status status DEFAULT 'pending'  
);
```

Create an Enum type. When creating the table, define a default value for the column that must be equal to one of the values in the enum type.



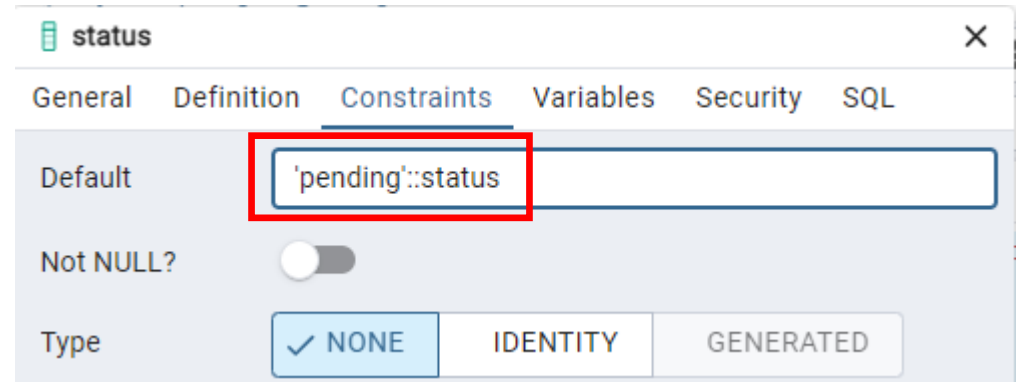
status

General Definition Constraints Variables Security SQL

Data type status

Length/Precision

Scale



status

General Definition Constraints Variables Security SQL

Default 'pending'::status

Not NULL? ☐

Type ☒ NONE ☐ IDENTITY ☐ GENERATED

```
INSERT INTO cron_jobs (cron_job_id) VALUES (1);  
INSERT INTO cron_jobs (cron_job_id) VALUES (2);
```

```
SELECT * FROM cron_jobs;
```

```
INSERT INTO cron_jobs (cron_job_id, status) VALUES (3, 'approved');
```

```
SELECT * FROM cron_jobs;
```

	cron_job_id integer	status status
1	1	pending
2	2	pending
3	3	approved

	cron_job_id integer	status status
1	1	pending
2	2	pending

7.15 Create a Type If Not Exists Using PL/pgSQL Function

```
DO
$$
BEGIN
    IF NOT EXISTS (SELECT *
                    FROM pg_type typ
                    INNER JOIN pg_namespace nsp
                    ON nsp.oid = typ.typnamespace
                    WHERE nsp.nspname = current_schema()
                    AND typ.typname = 'ai') THEN
        CREATE TYPE ai
            AS (a text,i integer);
    END IF;
END;
LANGUAGE postgresql;
```

This script will run and first check if a type called “ai” exists. If it does not exist it will create it. If it does already exist then it will do nothing.

8. PostgreSQL Constraints

[8.1 Constraints Introduction](#)

[8.2 Not Null Constraint](#)

[8.3 Unique Constraint](#)

[8.4 Default Constraint](#)

[8.5 Primary Key Constraint](#)

[8.6 Primary Key Constraints on Multiple Columns](#)

[8.7 Foreign Key Constraint](#)

[8.8 Tables Without Foreign Key Constraints](#)

[8.9 Create Foreign Key Constraint](#)

[8.10 Foreign Keys maintain referential data integrity](#)

[8.11 Drop a constraint](#)

[8.12 Add or Update Foreign Key Constraint on Existing Table](#)

[8.13 Add Check Constraint to New Table](#)

[8.14 Add, Rename, Drop Check Constraint on Existing table](#)

8.1 Constraints Introduction

Constraints are like gatekeepers that control the information that goes into a database

Constraints are the rules enforced on data columns in a table

These are used to prevent invalid data from being entered into the database

They ensure the accuracy and reliability of the data in the database

Constraints can be added on:

- Table
- Columns

Column level constraints are applied to only one column

Table level constraints are applied to whole table

Types of Constraints:

NOT NULL	Field must have values
UNIQUE	Only unique values are allowed
DEFAULT	Ability to set default values
PRIMARY KEY	Uniquely identifies each row/record in a database table
FOREIGN KEY	Constraints data based on columns in another table
CHECK	Checks all values meet specific criteria

8.2 Not Null Constraint

NULL represents unknown or information missing. NULL is not the same as an empty string or the number zero. To check if a value is NULL or not, use the IS NULL Boolean operator. Create not null on a table column as follows:

```
CREATE TABLE name (  
    Column _name data_type NOT NULL  
);
```

```
CREATE TABLE table_nn (  
    id SERIAL PRIMARY KEY,  
    tag TEXT NOT NULL  
);
```

```
INSERT INTO table_nn (tag)  
VALUES ('Elliott');
```

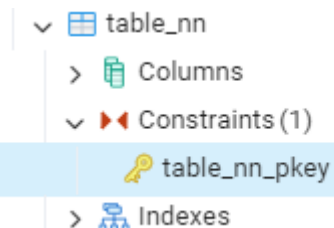
```
SELECT * FROM table_nn;
```

	id [PK] integer	tag text
1	1	Elliott

```
INSERT INTO table_nn (tag) VALUES (null);  
ERROR: Failing row contains (2, null).null value in column "tag" of relation  
"table_nn" violates not-null constraint
```

```
ERROR: null value in column "tag" of relation "table_nn" violates not-null  
constraint
```

```
SQL state: 23502 Detail: Failing row contains (2, null).
```



Note that I cannot insert a null value into the column because of the NOT NULL constraint. In PgAdmin 4 → Database → Table → Constraints, I can see the pkey (primary Key) constraint.

```
INSERT INTO table_nn (tag) VALUES ('null');
```

```
SELECT * FROM table_nn;
```

	id [PK] integer	tag text
1	1	Elliott
2	3	null

Interestingly, if I try and insert 'null' in quite marks as a string then it does get inserted, event though the column has a NOT NULL constraint.

```
INSERT INTO table_nn (tag) VALUES ('0');
```

```
SELECT * FROM table_nn;
```

	id [PK] integer	tag text
1	1	Elliott
2	3	null
3	4	0

Inserting zero as a string is added as zero. Zero is not a NULL but a value.

```
CREATE TABLE table_nn2 (  
    id serial PRIMARY KEY,  
    tag2 TEXT  
);
```

```
ALTER TABLE table_nn2  
ALTER COLUMN tag2 SET NOT NULL;
```

Add a NOT NULL constraint to an existing column.

8.3 Unique Constraint

Ensures that values stored in a column or group of columns are unique. Every time you insert row data it will check if the data for the unique column is truly unique. It will do the same if you try to update existing data in a unique column.

```
CREATE TABLE name (  
    Column _name data_type UNIQUE  
);
```

```
CREATE TABLE table_emails (  
    id SERIAL PRIMARY KEY,  
    email TEXT UNIQUE  
);
```

```
INSERT INTO table_emails (email)  
VALUES ('joeblack@gmail.com');
```

```
SELECT * FROM table_emails;
```

	id [PK] integer	email text
1	1	joeblack@gmail.com

```
INSERT INTO table_emails (email) VALUES ('joeblack@gmail.com');  
ERROR: Key (email)=(joeblack@gmail.com) already exists. duplicate key value  
violates unique constraint "table_emails_email_key"  
  
ERROR: duplicate key value violates unique constraint "table_emails_email_key"  
  
SQL state: 23505 Detail: Key (email)=(joeblack@gmail.com) already exists.
```

When you insert data into a column with a UNIQUE constraint it will first check all values in that column for all rows to see if that value already exists. If the value already exists it will throw an error and not insert the row.

```
CREATE TABLE table_products (  
    id SERIAL PRIMARY KEY,  
    product_code VARCHAR(10),  
    Product_name TEXT  
    UNIQUE (product_code, Product_name)  
);
```

If I am creating a table from new and I want to apply a UNIQUE constraint to multiple columns I would do it like this.

```
CREATE TABLE table_products (
  id SERIAL PRIMARY KEY,
  product_code VARCHAR(10),
  Product_name TEXT
);
```

However I can add a named constraint to multiple columns by altering the table. Note that the order of the columns is important.

```
ALTER TABLE table_products
ADD CONSTRAINT unique_products_code UNIQUE (product_code, product_name);
```

```
INSERT INTO table_products (product_code, product_name) VALUES ('A', 'Apple');
```

```
INSERT INTO table_products (product_code, product_name) VALUES ('A', 'Apple');
ERROR: Key (product_code, product_name)=(A, Apple) already exists.duplicate key value
violates unique constraint "unique_products_code"
ERROR: duplicate key value violates unique constraint "unique_products_code"
SQL state: 23505 Detail: Key (product_code, product_name)=(A, Apple) already exists.
```

```
INSERT INTO table_products (product_code, product_name) VALUES ('B', 'Apple');
```

	id [PK] integer	product_code character varying (10)	product_name text
1	1	A	Apple
2	3	B	Apple

However I can add a one column with the same value and one column that is different because I applied the unique constraint to both columns, both column values have to be the same to fail.

```
INSERT INTO table_products (product_code, product_name) VALUES ('Apple', 'A');
```

	id [PK] integer	product_code character varying (10)	product_name text
1	1	A	Apple
2	3	B	Apple
3	4	Apple	A

When the single constraint is applied to two columns the values are kind of concatenated and then checked for uniqueness.

8.4 Default Constraint

```
CREATE TABLE employees (  
  employee_id SERIAL PRIMARY KEY,  
  first_name VARCHAR(50),  
  last_name VARCHAR(50),  
  is_enable VARCHAR(2) DEFAULT 'Y'  
);
```

When columns are created without a default value explicitly set, then the default value will be NULL.

```
INSERT INTO employees (first_name,  
  last_name) VALUES ('JOHN', 'SMITH');
```

```
SELECT * FROM employees;
```

	employee_id [PK] integer	first_name character varying (50)	last_name character varying (50)	is_enable character varying (2)
1	2	JOHN	SMITH	Y

```
ALTER TABLE employees  
ALTER COLUMN is_enable SET DEFAULT 'N';
```

I can alter the column default value.

```
INSERT INTO employees (first_name,  
  last_name) VALUES ('JACK', 'JONES');
```

```
SELECT * FROM employees;
```

	employee_id [PK] integer	first_name character varying (50)	last_name character varying (50)	is_enable character varying (2)
1	2	JOHN	SMITH	Y
2	3	JACK	JONES	N

```
ALTER TABLE employees  
ALTER COLUMN is_enable DROP DEFAULT;
```

I can remove the default.

```
INSERT INTO employees (first_name,  
  last_name) VALUES ('JASON', 'BORNE');
```

```
SELECT * FROM employees;
```

	employee_id [PK] integer	first_name character varying (50)	last_name character varying (50)	is_enable character varying (2)
1	2	JOHN	SMITH	Y
2	3	JACK	JONES	N
3	5	JASON	BORNE	[null]

Set a default value of a column on a new table
<column> DEFAULT <value>;

SET a default value to an existing table

**ALTER TABLE tablename
ALTER COLUMN column_name SET DEFAULT;**

Drop a default value

**ALTER TABLE tablename
ALTER COLUMN column DROP DEFAULT;**

8.5 Primary Key Constraint

Uniquely identifies each record in a database table

There can be more UNIQUE columns, but only one primary key in a table

A primary key is a field in a table, which uniquely identifies each row/record in a database

Primary key must contain unique values

A table can only have one primary key, which may consist of a single or multiple fields

When multiple fields are used as a primary key, they are called a composite key

They are the same as UNIQUE NOT NULL

```
CREATE TABLE items (  
    item_id INTEGER PRIMARY KEY,  
    item_name VARCHAR(100) NOT NULL  
);
```

```
INSERT INTO items (item_id, item_name) VALUES (1, 'Pen');
```

```
SELECT * FROM items;
```

PK = Primary Key

item_id		item_name
[PK] integer		character varying (100)
1	1	Pen

Primary Key cannot be Null or a duplicate must be UNIQUE NOT NULL

```
INSERT INTO items (item_id, item_name) VALUES (1, 'Pen');  
ERROR: Key (item_id)=(1) already exists. duplicate key value violates unique constraint  
"items_pkey"  
ERROR: duplicate key value violates unique constraint "items_pkey"  
SQL state: 23505 Detail: Key (item_id)=(1) already exists.
```

```
INSERT INTO items (item_id, item_name) VALUES (null, 'Pen');  
ERROR: Failing row contains (null, Pen).null value in column "item_id" of relation  
"items" violates not-null constraint  
ERROR: null value in column "item_id" of relation "items" violates not-null constraint  
SQL state: 23502 Detail: Failing row contains (null, Pen).
```

```
ALTER TABLE items
DROP CONSTRAINT items_pkey;
```

I can drop a primary key constraint
using drop constraint tablename_pkey

```
ALTER TABLE items
ADD PRIMARY KEY (item_id);
```

I can use the add primary key command to re-add the PK.

8.6 Primary Key Constraints on Multiple Columns

```
ALTER TABLE items
ADD PRIMARY KEY (item_id, item_name);
```

I can add a PK that is based on two columns called
a **COMPOSITE PRIMARY KEY**

From Database → Constraints → Pkey I can
see that the PK is added across two columns.

items_pkey

General Definition SQL

Columns: item_id item_name

Include columns:

Note that the uniqueness is now spread across two columns so I can add the same values in one column as long as both columns are not the same as another row.

```
INSERT INTO items (item_id, item_name)
VALUES (2, 'Pen');
```

```
SELECT * FROM items;
```

	item_id [PK] integer	item_name [PK] character varying (100)
1	1	Pen
2	2	Pen

```
INSERT INTO items (item_id, item_name)
VALUES (2, '');
```

```
SELECT * FROM items;
```

	item_id [PK] integer	item_name [PK] character varying (100)
1	1	Pen
2	2	Pen
3	2	

```
CREATE TABLE t_grades (
  course_id VARCHAR(100) NOT NULL,
  student_id VARCHAR(100) NOT NULL,
  grade INT NOT NULL
);

INSERT INTO t_grades (course_id, student_id, grade) VALUES
('MATH', '52', 70),
('CHEMISTRY', '51', 70),
('ENGLISH', '52', 70),
('PYSYICS', '51', 70)
;
```

If I look at the table and it's data, none of the rows have a unique column but if I use a composite primary key then I can find uniqueness.

```
CREATE TABLE t_grades (
  course_id VARCHAR(100) NOT NULL,
  student_id VARCHAR(100) NOT NULL,
  grade INT NOT NULL,
  PRIMARY KEY (course_id, student_id)
);

INSERT INTO t_grades (course_id, student_id, grade) VALUES
('MATH', '52', 70),
('CHEMISTRY', '51', 70),
('ENGLISH', '52', 70),
('PYSYICS', '51', 70)
;
```

I can define the primary key like this and use two columns to obtain uniqueness.

	course_id [PK] character varying (100)	student_id [PK] character varying (100)	grade integer
1	MATH	52	70
2	CHEMISTRY	51	70
3	ENGLISH	52	70
4	PYSYICS	51	70

```
INSERT INTO t_grades (course_id, student_id, grade) VALUES ('MATH', '52', 67);
ERROR: Key (course_id, student_id)=(MATH, 52) already exists.duplicate key value
violates unique constraint "t_grades_pkey"
ERROR: duplicate key value violates unique constraint "t_grades_pkey"
SQL state: 23505 Detail: Key (course_id, student_id)=(MATH, 52) already exists.
```



```
ALTER TABLE t_grades  
DROP CONSTRAINT t_grades_pkey;
```

I can drop a primary key constraint
using drop constraint tablename_pkey

```
ALTER TABLE t_grades  
ADD CONSTRAINT t_grades_course_id_student_id_pkey  
PRIMARY KEY (course_id, student_id);
```

I can add a composite pkey by adding a
named constraint then defining the
column(s) that I want to use for the pkey.

t_grades_course_id_student_id_pkey

General Definition SQL

Name t_grades_course_id_student_id_pkey

Comment

t_grades_course_id_student_id_pkey

General Definition SQL

Columns course_id student_id

Include columns

8.7 Foreign Key Constraint

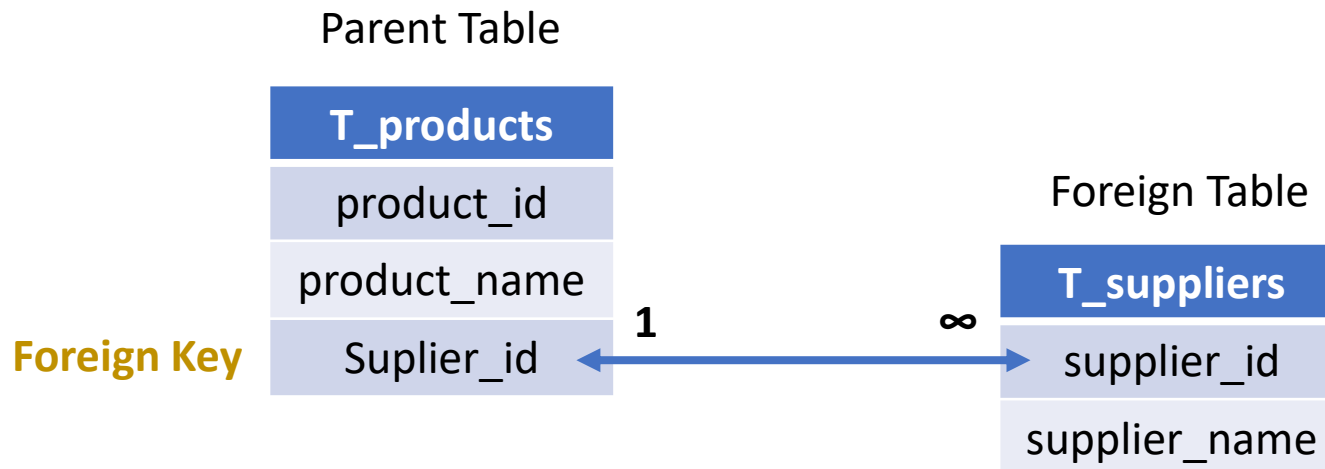
Foreign keys play a vital most important role in PostgreSQL

A foreign key is a column or group of columns in a table that references the primary key of another table.

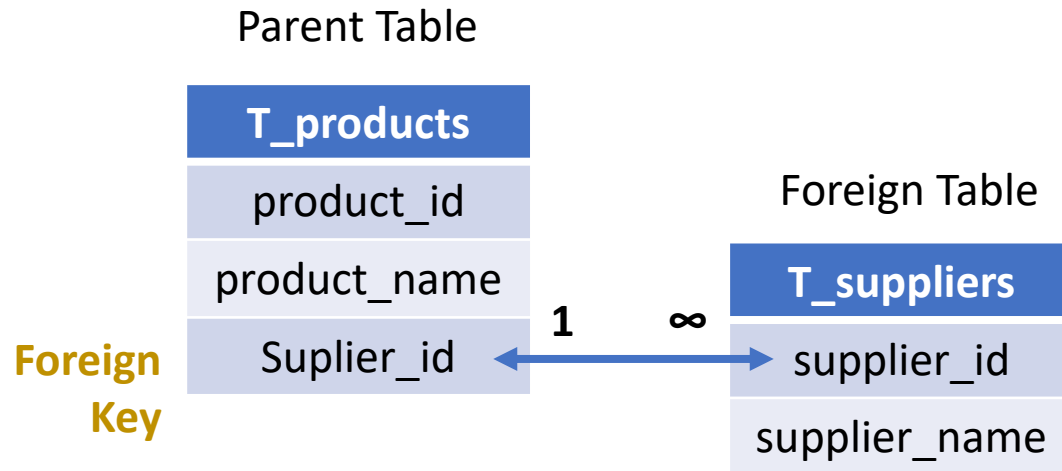
Parent table → Child/foreign table

Also in general words, foreign key in PostgreSQL is defined as the first table that has reference to the primary key of the second table

```
CREATE TABLE table_name (  
    column_name data_type PRIMARY KEY,  
    ...  
    ...  
    FOREIGN KEY (parent_column_name) REFERENCES child_table_name (child_column_name)  
);
```



8.8 Tables Without Foreign Key Constraints



```
CREATE TABLE t_products (  
    product_id INT PRIMARY KEY,  
    product_name VARCHAR(100) NOT NULL,  
    Supplier_id INT NOT NULL  
);  
  
CREATE TABLE t_suppliers (  
    supplier_id INT PRIMARY KEY,  
    supplier_name VARCHAR(100) NOT NULL  
);
```

```
INSERT INTO t_suppliers (supplier_id, supplier_name) VALUES  
(1, 'SUPPLIER 1'),  
(2, 'SUPPLIER 2');  
  
INSERT INTO t_products (product_id, product_name, supplier_id) VALUES  
(1, 'PEN', 1),  
(2, 'PAPER', 2);
```

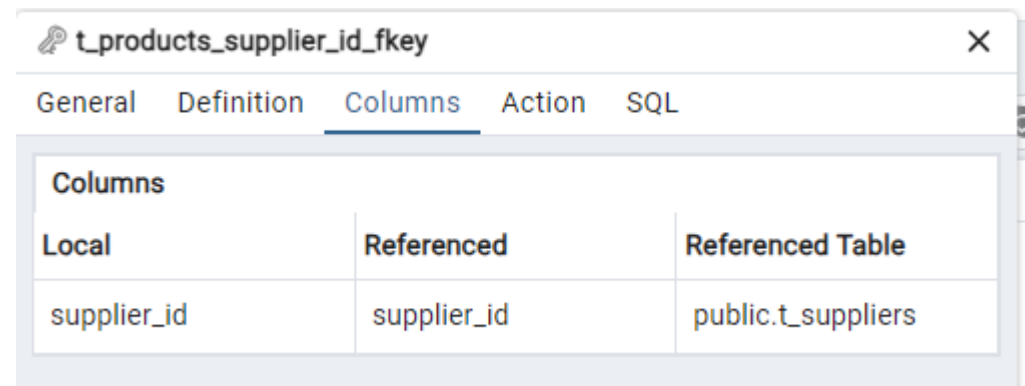
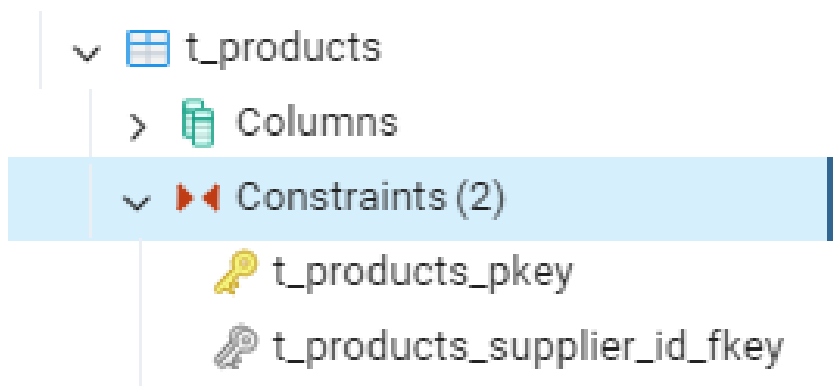
```
INSERT INTO t_products (product_id, product_name, supplier_id)  
VALUES (1, 'COMPUTER', 10);
```

Without a foreign key defined someone could add a new product to the table which does not have a supplier record which is a **LOSS OF DATA INTEGRITY**. Without correctly defined relationships the data will get corrupted and the link between product and supplier tables will break down with orphan products that do not have a supplier.

8.9 Create Foreign Key Constraint

```
CREATE TABLE t_suppliers (  
    supplier_id INT PRIMARY KEY,  
    supplier_name VARCHAR(100) NOT NULL  
);  
  
CREATE TABLE t_products (  
    product_id INT PRIMARY KEY,  
    product_name VARCHAR(100) NOT NULL,  
    Supplier_id INT NOT NULL,  
    FOREIGN KEY (supplier_id) REFERENCES t_suppliers (supplier_id)  
);
```

The child table contains the foreign key and the parent table references the child table.



8.10 Foreign Keys maintain referential data integrity

Foreign key constraints in PostgreSQL states that the values in the parent column must appear or present with values with a foreign column. If the value is not present in the foreign table it will show an error message (ERROR: insert or update on table "table name" violates foreign key constraint "table2 column_name fkey")

Foreign key is a type of constraint in PostgreSQL. Foreign key states that values in the column must match with values from some other row from another table.

```
INSERT INTO t_suppliers (supplier_id, supplier_name) VALUES
(1, 'SUPPLIER 1'),
(2, 'SUPPLIER 2');

INSERT INTO t_products (product_id, product_name, supplier_id) VALUES
(1, 'PEN',1),
(2, 'PAPER', 2);
```

```
INSERT INTO t_products (product_id, product_name, supplier_id) VALUES
(3, 'COMPUTER',10);
ERROR: Key (supplier_id)=(10) is not present in table "t_suppliers".insert or update
on table "t_products" violates foreign key constraint "t_products_supplier_id_fkey"
ERROR: insert or update on table "t_products" violates foreign key constraint
"t_products_supplier_id_fkey" SQL state: 23503 Detail: Key (supplier_id)=(10) is not
present in table "t_suppliers".
```

```
INSERT INTO t_suppliers (supplier_id, supplier_name) VALUES
(10, 'SUPPLIER 3');

INSERT INTO t_products (product_id, product_name, supplier_id) VALUES
(3, 'COMPUTER',10);
```

This works now
because the FK exists
in the child table

```
DELETE FROM t_suppliers WHERE supplier_id = 10;
ERROR: Key (supplier_id)=(10) is still referenced from table "t_products".update or
delete on table "t_suppliers" violates foreign key constraint
"t_products_supplier_id_fkey" on table "t_products"
ERROR: update or delete on table "t_suppliers" violates foreign key constraint
"t_products_supplier_id_fkey" on table "t_products"
SQL state: 23503 Detail: Key (supplier_id)=(10) is still referenced from table
"t_products".
```

```
DELETE FROM t_products WHERE product_id = 3;
DELETE FROM t-supplier WHERE supplier_id = 10;
```

Now I can delete a row from a child table because I have deleted the referencing row from the parent table first.

```
UPDATE t_products SET supplier_id = 2 WHERE product_id = 1;
```

I can update a parent table row to reference to a different child table row.

```
UPDATE t_products SET supplier_id = 100 WHERE product_id = 1;
ERROR: Key (supplier_id)=(100) is not present in table "t_suppliers".insert or
update on table "t_products" violates foreign key constraint
"t_products_supplier_id_fkey"
ERROR: insert or update on table "t_products" violates foreign key constraint
"t_products_supplier_id_fkey"
SQL state: 23503 Detail: Key (supplier_id)=(100) is not present in table
"t_suppliers".
```

I cannot update the parent table row to reference a row in the child table that does not exist.

8.11 Drop a constraint

```
ALTER TABLE t_products  
DROP CONSTRAINT t_products_supplier_id_fkey;
```

Use this SQL command to drop a FK constraint.

8.12 Add or Update Foreign Key Constraint on Existing Table

To add or update a FK constraint use the syntax:

```
ALTER TABLE t_products  
ADD CONSTRAINT fk_name (parent_table_col) REFERENCES child_table (child_table_col);
```

```
ALTER TABLE t_products  
ADD CONSTRAINT t_products_supplier_id_fk FOREIGN KEY (supplier_id) REFERENCES t_supplier  
(supplier_id);
```

8.12 Check Constraint

A check constraint is a kind of constraint that allows you to specify **if values in a column must meet a specific requirement**.

The Check constraint uses a **Boolean expression** to evaluate the values before they are **inserted or updated** to the column.

If the values pass the check, PostgreSQL will INSERT or UPDATE these values to the column. Otherwise, PostgreSQL will reject the changes and issue a constraint violation error.

8.13 Add Check Constraint to New Table

```
CREATE TABLE staff (  
  staff_id SERIAL PRIMARY KEY,  
  first_name VARCHAR(50),  
  last_name VARCHAR(50),  
  birth_date DATE CHECK (birth_date > '1900-01-01'),  
  joined_date DATE CHECK (joined_date > birth_date),  
  salary NUMERIC CHECK (salary > 0)  
);
```

The Check constraint is to verify that birth date is after 1st January 1900, That the employee joined the company on a date that is after their birth and that their salary is a positive number.

```
INSERT INTO staff (first_name, last_name, birth_date, joined_date, salary)  
VALUES ('Elliott', 'WARD', '1989-10-20', '2002-01-05', 1000000)
```



```

INSERT INTO staff (first_name, last_name, birth_date, joined_date, salary)
VALUES ('John', 'SMITH', '1990-10-20', '1990-09-20', 10000);
ERROR: Failing row contains (2, John, SMITH, 1990-10-20, 1990-09-20, 10000).new row
for relation "staff" violates check constraint "staff_check"
ERROR: new row for relation "staff" violates check constraint "staff_check"
SQL state: 23514 Detail: Failing row contains (2, John, SMITH, 1990-10-20, 1990-09-20,
10000).

```

```

INSERT INTO staff (first_name, last_name, birth_date, joined_date, salary)
VALUES ('John', 'SMITH', '1990-10-20', '2009-09-20', 10000);

```

	staff_id [PK] integer	first_name character varying (50)	last_name character varying (50)	birth_date date	joined_date date	salary numeric
1	1	Elliott	WARD	1989-10-20	2002-01-05	1000000
2	4	John	SMITH	1990-10-20	2009-09-20	10000

```

UPDATE Staff
SET salary = 0
WHERE staff_id = 4;
ERROR: Failing row contains (4, John, SMITH, 1990-10-20, 2009-09-20, 0).new row
for relation "staff" violates check constraint "staff_salary_check"
ERROR: new row for relation "staff" violates check constraint
"staff_salary_check"
SQL state: 23514 Detail: Failing row contains (4, John, SMITH, 1990-10-20, 2009-
09-20, 0).

```

8.14 Add, Rename, Drop Check Constraint on Existing table

```
CREATE TABLE prices (  
  price_id SERIAL PRIMARY KEY,  
  product_id INT NOT NULL,  
  price NUMERIC NOT NULL,  
  discount NUMERIC NOT NULL,  
  Valid_from DATE NOT NULL,  
);
```

I want to add constraints to the table so that:

1. Price is greater than zero
2. Discount >= zero
3. Price > discount

```
ALTER TABLE prices  
ADD CONSTRAINT price_check  
CHECK (  
  price > 0  
  AND discount >= 0  
  AND price > discount  
);
```

```
INSERT INTO prices (product_id, price, discount, valid_from) VALUES (1, 100, 20, '2020-10-01');
```

```
INSERT INTO prices (product_id, price, discount, valid_from) VALUES (2, 100, 120, '2020-10-01');  
ERROR: Failing row contains (2, 2, 100, 120, 2020-10-01).new row for relation "prices"  
violates check constraint "price_check"  
ERROR: new row for relation "prices" violates check constraint "price_check"  
SQL state: 23514 Detail: Failing row contains (2, 2, 100, 120, 2020-10-01).
```

```
ALTER TABLE prices  
RENAME CONSTRAINT price_check TO price_discount_check;
```

Rename a constraint. Use the name of the constraint.

```
ALTER TABLE prices  
DROP CONSTRAINT price_discount_check;
```

Drop a constraint using it's name

9. PostgreSQL Sequences

[9.1 Create a Sequence, Advance a Sequence, Get current value, Set Value](#)

[9.2 Restart, rename a sequence, and use PgAdmin 4 to alter a sequence](#)

[9.3 Create a Sequence with Start with, increment, MinValue & MaxValue](#)

[9.4 Create a Sequence Using a Particular Data Type](#)

[9.4 Create a Descending Sequence & Cycle it](#)

[9.5 Delete a sequence](#)

[9.6 Attach a Sequence to Table Column](#)

[9.7 List all Sequences in a Database](#)

[9.8 Share One Sequence Between Two Tables](#)

[9.9 Create an Alphanumeric Sequence](#)

9.1 Create a Sequence, Advance a Sequence, Get current value, Set Value

```
CREATE SEQUENCE IF NOT EXIST test_seq
```

```
SELECT setval('test_seq', 100);  
SELECT nextval('test_seq');
```

	nextval bigint
1	101

I can manually set the sequence value to 100 which would make the next value 101.

```
SELECT setval('test_seq', 200, false);  
SELECT currval('test_seq'),nextval('test_seq');
```

	currval bigint	nextval bigint
1	101	200

If I set value and add a false flag, what this does is say that the next value will be the same as the set value. Note how current value is set to 101 from previously.

```
CREATE SEQUENCE IF NOT EXIST test_seq2 START WITH 100;
```

Create a sequence with a specific start value

Show the next value and current value of the sequence.

```
SELECT nextval('test_seq');
```

```
SELECT currval('test_seq');
```

9.2 Restart, rename a sequence, and use PgAdmin 4 to alter a sequence

```
ALTER SEQUENCE test_seq RESTART WITH 100;  
SELECT nextval ('test_seq');
```

	nextval bigint
1	100

I can restart a sequence with a new value

```
ALTER SEQUENCE test_seq RENAME TO my_sequence_4;
```

I can rename a sequence and also view and alter any of its values in pgAdmin 4.

From PgAdmin 4 → Database_name → Sequences
→ right click → properties

1.3 my_sequence_4

General Definition Security SQL

Current value 100

Increment 1

Start 1

Minimum 1

Maximum 9223372036854775807

Cache 1

Cycled ☐

```
ALTER SEQUENCE IF EXISTS public.my_sequence_4 INCREMENT 10;
```

I can set a sequence to increment by a value that is not the default of 1, for example increment by 10.

9.3 Create a Sequence with Start with, increment, MinValue and MaxValue

```
CREATE SEQUENCE IF NOT EXISTS  
test_seq3  
INCREMENT 50  
MINVALUE 400  
MAXVALUE 6000  
START WITH 5000;
```

```
SELECT nextval('test_seq3');
```

The start value (start with) is 500

	nextval bigint
1	5000

```
SELECT nextval('test_seq3');
```

If I run it again I see that the sequence is incrementing by 50

	nextval bigint
1	5050

```
SELECT nextval('test_seq3');
```

The sequence increments by 50 each time it is run

	nextval bigint
1	5100

```
SELECT nextval('test_seq3');
```

I run the sequence until it almost reaches its MaxValue

	nextval bigint
1	5950

```
SELECT nextval('test_seq3');
```

The sequence is at its MaxValue

	nextval bigint
1	6000

```
SELECT nextval('test_seq3');  
ERROR: nextval: reached maximum value  
of sequence "test_seq3" (6000) SQL  
state: 2200H
```

Once the sequence reaches its max value it will no longer run.

9.4 Create a Sequence Using a Particular Data Type

By default a sequence is created with a data type of BIGINT but it can be manually configured as SMALLINT | INT | BIGINT. This may sometimes be necessary to optimise a database and save space. Use the AS keyword to set the datatype.

```
CREATE SEQUENCE IF NOT EXISTS test_seq_smallint AS SMALLINT;  
CREATE SEQUENCE IF NOT EXISTS test_seq_int AS INT;  
CREATE SEQUENCE IF NOT EXISTS test_seq4;
```

9.4 Create a Descending Sequence & Cycle it

```
CREATE SEQUENCE IF NOT EXISTS test_seq5  
INCREMENT -1  
MINVALUE 1  
MAXVALUE 3  
START WITH 3  
CYCLE;
```

This sequence will start at 3 and count down to 1.

The CYCLE keyword will reset the query automatically. I can equally specify NO CYCLE.

Note how this sequence starts at 3 then cycles down to 1 and resets back to 1.

A use case of a cycling sequence would be monthly coupons whereby a limited quantity of 100 are issued every month.

```
SELECT nextval('test_seq5');
```

	nextval bigint 
1	3

```
SELECT nextval('test_seq5');
```

	nextval bigint 
1	1

```
SELECT nextval('test_seq5');
```

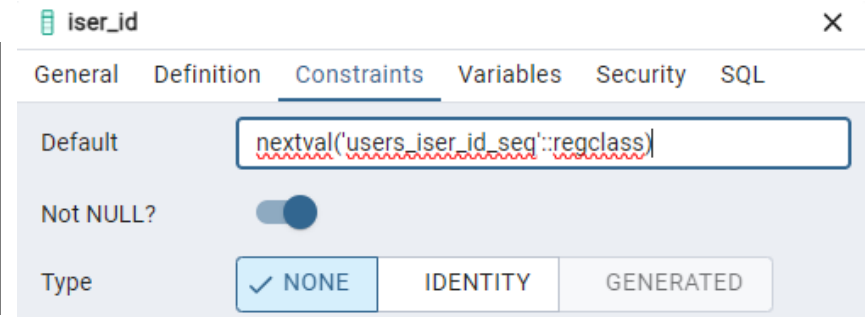
	nextval bigint 
1	3

9.5 Delete a Sequence

```
DROP SEQUENCE test_seq2; Use DROP SEQUENCE seq_name;
```

9.6 Attach a Sequence to Table Column

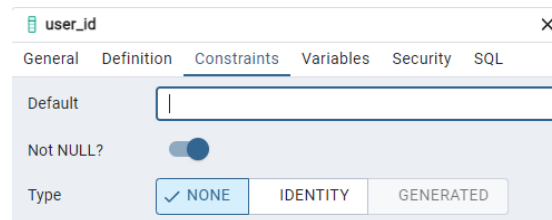
```
CREATE TABLE users (  
    user_id SERIAL PRIMARY KEY,  
    user_name VARCHAR(50)  
);  
  
INSERT INTO users (user_name) VALUES ('Elliott');  
INSERT INTO users (user_name) VALUES ('Elliott2');
```



The screenshot shows the 'Constraints' tab for the 'user_id' column. The 'Default' field is populated with the SQL expression `nextval('users_user_id_seq'::regclass)`. The 'Not NULL?' checkbox is checked. The 'Type' dropdown is set to 'NONE'.

By default this table has a sequence attached to the `user_id` because it is created with the type `SERIAL` which is an `INT` with sequencing. The sequence can be seen under the column properties in PgAdmin 4 where we see `nextval` in the default setting.

```
CREATE TABLE users2 (  
    user_id INT PRIMARY KEY,  
    user_name VARCHAR(50)  
);
```



The screenshot shows the 'Constraints' tab for the 'user_id' column. The 'Default' field is empty. The 'Not NULL?' checkbox is checked. The 'Type' dropdown is set to 'NONE'.

If I create a table column as `INT`, it does not have a sequence attached. Note how the default column is empty.

```
CREATE SEQUENCE users2_user_id_seq  
START WITH 100 OWNED BY users2.user_id;
```

I can create a sequence named `table_column_seq` and set it to start with 100. I have to say what `table.column` owns the sequence.

```
ALTER TABLE users2  
ALTER COLUMN user_id SET DEFAULT nextval('users2_user_id_seq');
```

Alter the table so that the column uses the sequence `nextval` as a default value.


```
INSERT INTO users2 (user_name) VALUES ('Elliott');
INSERT INTO users2 (user_name) VALUES ('Elliott2');
SELECT * FROM users2;
```

Now I see that the sequence is attached and starting from 100

	user_id [PK] integer	user_name character varying (50)
1	100	Elliott
2	101	Elliott2

9.7 List all Sequences in a Database

```
SELECT relname sequence_name
FROM pg_class
WHERE relkind = 'S';
```

This query will show all sequences in a database.

pg_class is a table that contains all the sequences.

	sequence_name name
5	my_sequence_4
6	sample_sample_id_seq
7	addresses_address_id_seq
8	client_names_client_name_id_seq
9	companies_comp_id_seq
10	test_seq3
11	test_seq_smallint
12	test_seq_int
13	test_seq4
14	test_seq5
15	users_user_id_seq

The screenshot shows the PgAdmin 4 interface. On the left, a tree view shows the database structure: 'Publications' (expanded), 'Schemas (2)' (expanded), and 'public' (expanded). Under 'public', several objects are listed: 'Aggregates', 'Collations', 'Domains', 'FTS Configurations', 'FTS Dictionaries', 'FTS Parsers', 'FTS Templates', 'Foreign Tables', 'Functions', 'Materialized Views', 'Operators', 'Procedures', and 'Sequences (24)'. The 'Sequences (24)' folder is expanded, showing a list of sequences: '1.3 addresses_address_id_seq', '1.3 client_names_client_name_id_seq', '1.3 companies_comp_id_seq', and '1.3 employees_employee_id_seq'.

The sequences can also be seen in PgAdmin 4.
Database → Schema → public → sequences.

9.8 Share One Sequence Between Two Tables

```
CREATE SEQUENCE common_fruits_seq START WITH 100;

CREATE TABLE apples (
  fruit_id INT DEFAULT nextval('common_fruits_seq'),
  fruit_name VARCHAR(50)
);

CREATE TABLE mangoes (
  fruit_id INT DEFAULT nextval('common_fruits_seq'),
  fruit_name VARCHAR(50)
);
```

I can create a common sequence and two tables that each use the same common sequence to increment a column.

```
INSERT INTO apples (fruit_name) VALUES ('big_apple');
SELECT * from apples;
```

	fruit_id integer	fruit_name character varying (50)
1	100	big_apple

```
INSERT INTO mangoes (fruit_name) VALUES ('big_mango');
SELECT * from mangoes;
```

	fruit_id integer	fruit_name character varying (50)
1	101	big_mango

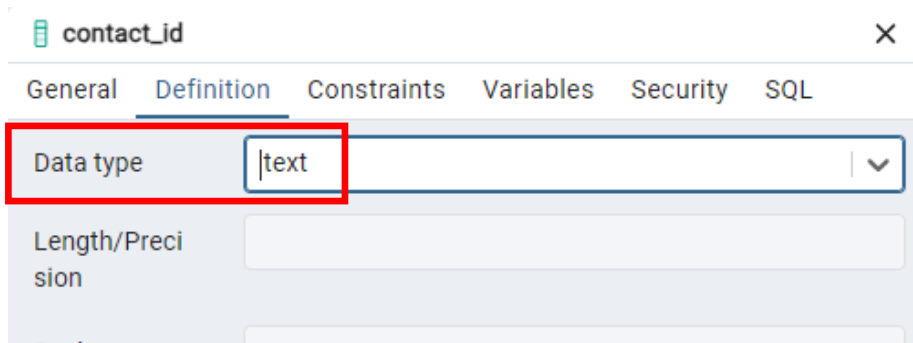
Note how both tables are sharing the same sequence because the fruit_id column is starting at 100.

9.9 Create an Alphanumeric Sequence

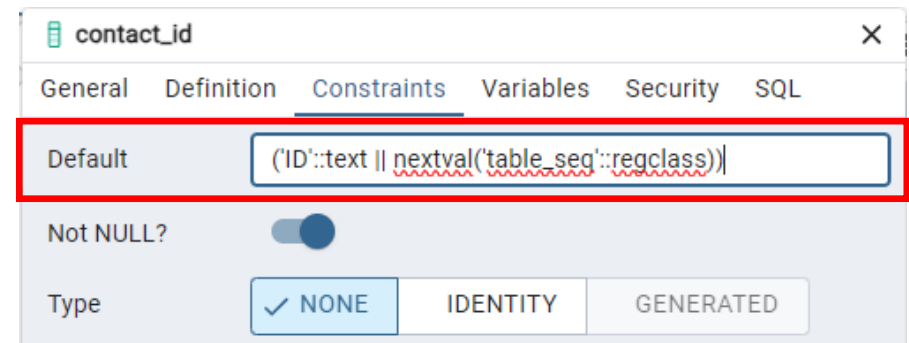
```
CREATE SEQUENCE table_seq;  
  
CREATE TABLE contacts (  
    contact_id TEXT NOT NULL DEFAULT ('ID' || nextval('table_seq')),  
    contact_name VARCHAR(150)  
);
```

By default, sequences are only numeric.

However I can prepend to a numeric number a text string by defining the column data type as text, not numeric (Smallint | int | bigint). I then define it as not null and set the default to contain an expression made of a string 'ID' piped to the nextval set to the sequence.



The screenshot shows the 'contact_id' column definition. The 'Data type' is set to 'text'. The 'Length/Precision' is empty. The 'Not NULL?' checkbox is checked. The 'Type' is set to 'NONE'.



The screenshot shows the 'contact_id' column constraints. The 'Default' is set to ('ID'::text || nextval('table_seq'::regclass)). The 'Not NULL?' checkbox is checked. The 'Type' is set to 'NONE'.

```
ALTER SEQUENCE table_seq OWNED BY contacts.contact_id;
```

Alter the sequence to be owned by the table.column_name

```
INSERT INTO contacts (contact_name) VALUES ('JACK');  
INSERT INTO contacts (contact_name) VALUES ('JOHN');  
SELECT * FROM contacts;
```

	contact_id text	contact_name character varying (150)
1	ID1	JACK
2	ID2	JOHN

Note how the id column is now a custom alphanumeric incrementing sequence.

10. String Functions

[10.1 Upper, Lower, IntCap](#)

[10.2 Left & Right](#)

[10.3 Reverse](#)

[10.4 Split Part](#)

[10.5 Trim, bTrim, lTrim, rTrim](#)

[10.6 Lpad & Rpad](#)

[10.7 Length](#)

[10.8 Position](#)

[10.9 Substring](#)

[10.10 Repeat](#)

[10.11 Replace](#)

10.1 Upper, Lower, IntCap

```
SELECT
  UPPER (first_name) AS FIRSTNAME,
  UPPER (last_name) AS LASTNAME
FROM directors;
```

UPPER can be used to uppercase all letters in a string.

	firstname text	lastname text
1	TOMAS	ALFREDSON
2	PAUL	ANDERSON
3	WES	ANDERSON
4	RICHARD	AYOADE

```
SELECT
  LOWER (first_name) AS FIRSTNAME,
  LOWER (last_name) AS LASTNAME
FROM directors;
```

LOWER can be used to lower case all letters in a string.

	firstname text	lastname text
1	tomas	alfredson
2	paul	anderson
3	wes	anderson
4	richard	ayoade

```
SELECT INITCAP ('The world is changing at lightning pace');
```

INITCAP will capitalise the first letter of each word in a string.

	initcap text
1	The World Is Changing At Lightning Pace

```
SELECT
  INITCAP
    (CONCAT(
      first_name, ' ', lastname
    ) AS name
  )
FROM directors
ORDER BY first_name;
```

	name text
1	Ang Lee
2	Bruce Lee
3	Chan-Wook Park
4	Fernando Meirelles
5	Florian Henckel Von Donnersmar...

I can combine INITCAT with CONCAT to make a new column.

10.2 Left & Right

```
SELECT  
LEFT (first_name,3) AS FIRSTNAME,  
LEFT (last_name, 3) AS LASTNAME  
FROM directors;
```

	firstname text	lastname text
1	mas	son
2	aul	son
3	Wes	son
4	ard	ade

	firstname text	lastname text
1	Tom	Alf
2	Pau	And
3	Wes	And
4	Ric	Ayo

The left function returns the first n number of characters from the left side of the string

```
SELECT  
RIGHT (first_name,3) AS FIRSTNAME,  
RIGHT (last_name, 3) AS LASTNAME  
FROM directors;
```

The right function returns the last n number of characters from the right side of the string

```
SELECT  
LEFT (first_name,-2) AS FIRSTNAME,  
LEFT (last_name, -2) AS LASTNAME  
FROM directors;
```

	firstname text	lastname text
1	mas	fredson
2	ul	derson
3	s	derson
4	chard	oade

	firstname text	lastname text
1	Tom	Alfreds
2	Pa	Anders
3	W	Anders
4	Richa	Ayoa

If I use a negative number then it will get all the characters from the left side minus 2.

```
SELECT  
RIGHT (first_name,-2) AS FIRSTNAME,  
RIGHT (last_name, -2) AS LASTNAME  
FROM directors;
```

A negative number for the right side will get all of the characters from the right minus 2, i.e the first two characters of the string.

List all initial letter's of the first name.

```
SELECT
  LEFT (first_name, 1)
  AS initial
FROM directors
ORDER by 1;
```

	initial text
1	A
2	B
3	C
4	F
5	F

Count the number of directors who's first name starts with A, B, C etc...

```
SELECT
  LEFT (first_name, 1)
  AS initial
  COUNT (*) as total_initials
FROM directors
GROUP by 1;
ORDER by 1;
```

	initial text	total_initials bigint
1	A	1
2	B	1
3	C	1
4	F	3
5	G	2
6	H	1
7	J	3
8	K	2
9	L	1
10	M	3
11	P	2

	movie_name character varying (100)	left text
1	A Clockwork Orange	A Cloc
2	Apocalypse Now	Apoc...
3	Battle Royale	Battle
4	Blade Runner	Blade
5	Chungking Express	Chun...
6	City of God	City o
7	City of Men	City o
8	Cold Fish	Cold F
9	Crouching Tiger Hidden Dragon	Crou...
10	Eyes Wide Shut	Eyes ...
11	Forrest Gump	Forres

Get the first six characters of every movie name

```
SELECT
  movie_name,
  LEFT (movie_name,6)
FROM movies;
```

Find all directors who's last name ends with 'on'

```
SELECT
  last_name
FROM directors
WHERE RIGHT (last_name, 2) = 'on';
```

	last_name character varying (150)
1	Alfredson
2	Anderson
3	Anderson
4	Besson
5	Cameron
6	Stevenson

10.3 Reverse

```
SELECT REVERSE ('Amazing PostGreSQL');
```

	reverse text
1	LQSerGtsoP gnizamA

The Reverse function returns the input string with all characters in reverse order.

```
SELECT REVERSE ('LQSerGtsoP ginzamA');
```

	reverse text
1	Amaznig PostGreSQL

10.4 Split_Part

Split_part function splits a string on a specific delimiter and returns the nth substring where the syntax is: **SPLIT_PART(string, delimiter, position)**

```
SELECT SPLIT_PART('1,2,3',',', 3);
```

	split_part text
1	3

Split the string '1,2,3' into an array on the coma and return the third part.

```
SELECT SPLIT_PART('A|B|C|D','|', 3);
```

	split_part text
1	C

Split the string 'A|B|C|D' into an array on the pipe and return the third part.

```
SELECT  
movie_name,  
release_date,  
SPLIT_PART(release_date,'-',1) AS release_year  
FROM movies;
```

```
ERROR: function split_part(date, unknown, integer) does not exist  
LINE 4: SPLIT_PART(release_date,'-',1) AS release_year  
          ^
```

HINT: No function matches the given name and argument types. You might need to add explicit type casts.

SQL state: 42883 Character: 37

Split_Part is a STRING function and release_date is a date type.

```
SELECT  
movie_name,  
release_date,  
SPLIT_PART(release_date::TEXT,'-',1)  
AS release_year  
FROM movies;
```

	movie_name character varying (100)	release_date date	release_year text
1	A Clockwork Orange	1972-02-02	1972
2	Apocalypse Now	1979-08-15	1979
3	Battle Royale	2001-01-04	2001

When I cast a date data type to a text type, I can use string functions on it.

10.5 Trim, bTrim, lTrim, rTrim

TRIM Removes the longest string that contains a specific character

LTRIM Removes all characters, spaces by default, from the beginning (left side) of a string

RTRIM Removes all characters, spaces by default, from the end (right side) of a string

BTRIM is the combination of LTRIM() and RTRIM functions

The syntax is: TRIM([LEADING | TRAILING | BOTH] [characters] FROM string);

TRIM(LEADING FROM string)

TRIM TRAILING FROM string)

TRIM BOTH FROM string)

LTRIM(string, [characters])

RTRIM(string, [characters])

BTRIM(string, [characters])

```
SELECT TRIM (LEADING FROM 'Amazing PostgreSQL');
```

	rtrim text
1	Amazing PostgreSQL

	ltrim text
1	Amazing PostgreSQL

```
SELECT TRIM (TRAILING FROM 'Amazing PostgreSQL');
```

```
SELECT TRIM ('Amazing PostgreSQL');
```

	btrim text
1	Amazing PostgreSQL

```
SELECT TRIM (LEADING '0' FROM CAST(00000123456 AS TEXT));
```

	ltrim text
1	123456

By default trim will remove leading and/or trailing spaces but I can also specify a character that I want to remove. For example a serial number that comes with a lot of leading zeros.

```
SELECT LTRIM('yummy', 'y');
```

```
SELECT BTRIM('yummy', 'y');
```

	ltrim text
1	ummy

	btrim text
1	umm

```
SELECT RTRIM('yummy', 'y');
```

	rtrim text
1	yumm

10.6 Lpad & Rpad

LPAD Function pads a string on the left to a specified length with a sequence of characters

RPAD Function pads a string on the right to a specified length with a sequence of characters

By default the pad is spaces unless a character is specified

```
SELECT LPAD('Database', 15, '*');
```

	lpad text
1	*****Database

```
SELECT RPAD('Database', 15, '*');
```

	rpad text
1	Database*****

```
SELECT RPAD('1111', 6, 'A');
```

	rpad text
1	1111AA

```
SELECT
  mv.movie_name,
  r.revenues_domestic,
  LPAD('*', CAST(TRUNC(r.revenues_domestic / 10) AS INT), '*') chart
FROM movies mv
INNER JOIN movies_revenues r ON r.movie_id = mv.movie_id
ORDER BY 3 DESC
NULLS LAST;
```

	movie_name character varying (100)	revenues_domestic numeric (10,2)	chart text
1	Titanic	659.20	*****
2	Star Wars: A New Hope	461.20	*****
3	Spider-Man	404.10	*****
4	Spider-Man 2	374.10	*****
5	Forrest Gump	330.30	*****
6	Spider-Man 3	337.00	*****

I am representing the movie revenue as a stars chart based on the highest grossing films.

10.7 Length

```
SELECT LENGTH('Amazing PostgreSQL');
```

	length integer
1	18

Returns the number of characters or bytes in a string.

```
SELECT LENGTH(CAST('123456' AS TEXT));
```

	length integer
1	6

I can cast a numeric data type as a text to use the length function.

```
SELECT LENGTH('What is the length of this string');
```

	length integer
1	33

Length includes space characters.

```
SELECT char_length('');
```

	char_length integer
1	0

Char_length produces the same result as LENGTH

```
SELECT char_length(' ');
```

	char_length integer
1	1

Spaces are counted as 1 character

```
SELECT char_length(NULL);
```

	char_length integer
1	[null]

Null values cannot have a length and will return null.

```
SELECT
  first_name || ' ' || last_name AS full_name,
  LENGTH(first_name || ' ' || last_name) full_name_length
FROM directors
ORDER BY 2 DESC;
```

Get directors with longest names

	full_name text	full_name_length integer
1	Florian Henckel von Donnersmar...	33
2	Francis Ford Coppola	20
3	Fernando Meirelles	18

10.8 Position

PostgreSQL POSITION() function returns the location of a substring in a string

Syntax: POSITION(substring IN String)

Returns an integer that represents to location of the substring within a string

Returns the first instance of the substring in the string

Searches for the substring case-sensitively

```
SELECT POSITION('Amazing' IN 'Amazing PostgreSQL');
```

	position integer
1	1

Starts counting from 1 and not zero.

```
SELECT POSITION('is' IN 'This is a computer');
```

	position integer
1	3

The first occurrence of 'is' is in the word 'This' so the third position is returned.

```
SELECT POSITION('A' IN 'KlickAnalytics.com');
```

	position integer
1	6

Position is case sensitive so it will find the uppercase A.

```
SELECT POSITION('C' IN 'KlickAnalytics.com');
```

	position integer
1	0

An uppercase C is not present in this string so position will return zero.

10.8 Strpos

Function to find the position, from where a substring is being matched within the string. The syntax is **STRPOS(<string>, <substring>)**

```
SELECT STRPOS('world Bank', 'Bank');
```

	strpos integer
1	7

	strpos integer
1	0

```
SELECT STRPOS('world Bank', 'bank');
```

Strpos is case sensitive as per position

```
SELECT
    first_name,
    last_name
FROM directors
WHERE STRPOS(last_name, 'on') > 0;
```

	first_name character varying (150)	last_name character varying (150)
1	Tomas	Alfredson
2	Paul	Anderson
3	Wes	Anderson
4	Luc	Besson

Find the first name and last name of all directors where there surname contains 'on'.

Difference between Position and strpos

These functions do exactly the same thing and only differ in syntax. Documentation for strops() says: Location of specific substring (same as position(substring, string) but note the reversed argument order.

The reason that they both exist and differ only in syntax is that POSITION(str1 IN str2) is defined by ANSI SQL standard.

10.9 Substring

Function to allow extraction of a substring from a string

Syntax: **SUBSTRING(string [from start position] for [length])**

or SUBSTRING(string, start position, length)

The first position in a string always starts from 1

	substring text
1	wonderful

```
SELECT SUBSTRING('What a wonderful world' FROM 1 FOR 4);
```

	substring text
1	What

```
SELECT SUBSTRING('What a wonderful world' FROM 8 FOR 10);
```

```
SELECT SUBSTRING('What a wonderful world' FOR 7);
```

	substring text
1	What a

```
SELECT
  first_name,
  last_name,
  SUBSTRING(first_name, 1, 1) AS initial
FROM directors
ORDER BY last_name;
```

	first_name character varying (150)	last_name character varying (150)	initial text
1	Tomas	Alfredson	T
2	Paul	Anderson	P
3	Wes	Anderson	W
4	Richard	Ayoade	R

Note that the data type of the output of SUBSTRING() is TEXT

10.10 Repeat

Function that repeats a string a specified number of times. Syntax:

REPEAT(string, n)

	repeat text
1	

```
SELECT REPEAT('A', 4);
```

	repeat text
1	AAAA

```
SELECT REPEAT(' ', 10);
```

```
SELECT REPEAT('AB', 10);
```

	repeat text
1	ABABABABABABABABABAB

10.11 Replace

Function that replaces ALL occurrences of a specified string. Replace is case sensitive. Syntax:

REPLACE(string, from_substring, to_substring)

	replace text
1	I like dogs

```
SELECT REPLACE('ABC XYZ', 'X', '1');
```

	replace text
1	ABC 1YZ

```
SELECT REPLACE('I like cats', 'cats', 'dogs');
```

```
SELECT REPLACE('What a wonderful world', 'a', 'an amazing');
```

	replace text
1	Whan amazingt an amazing wonderful world

	replace text
1	11144433

```
SELECT REPLACE('11122233', '2', '4');
```

Note all occurrences of '2' have been replaced with '4'

11. Aggregate Functions

[11.1 Counting Results via Count Function](#)

[11.2 Sum with Sum Function](#)

[11.3 Min & Max Functions](#)

[11.3 Greatest & Least Functions](#)

[11.4 GREATEST\(\) vs MAX\(\)](#)

[11.5 Average with AVG function](#)

[11.6 Combining Columns using Mathematical Operators](#)

11.1 Counting Results via Count Function

Syntax: `SELECT COUNT(*) FROM table_name` or `SELECT COUNT(Column_name) FROM table_name`

```
SELECT COUNT(*) FROM movies;
```

	count bigint
1	53

```
SELECT COUNT(movie_id) FROM movies;
```

	count bigint
1	53

```
SELECT COUNT(movie_lang) FROM movies;
```

	count bigint
1	53

Using distinct to count the number of movie languages

```
SELECT COUNT(DISTINCT(movie_lang)) FROM movies;
```

	count bigint
1	8

Using distinct to count the number of movie directors

```
SELECT COUNT(DISTINCT(director_id)) FROM movies;
```

	count bigint
1	37

```
SELECT COUNT(*) FROM movies  
WHERE movie_lang = 'English';
```

	count bigint
1	38

Count the number of English movies

```
SELECT COUNT(*) FROM movies  
WHERE movie_lang = 'Japanese';
```

	count bigint
1	4

Count the number of Japanese movies

Use either **COUNT(field)** or **COUNT(*)** and stick with it consistently

Use COUNT(*) for counting

For counting rows we could have this: **COUNT(emp) FROM emp;**

Counting with NULL

It is not good practice to make a field nullable if it's name matches the table name. For example values 'Apple', 'Banana', NULL, 'Pears' on a fruit field. This will not count all rows, it will yield 3, not 4. **SELECT COUNT(fruit) FROM fruit**

COUNT(1): The one trick pony

In particular to COUNT(1), it is a one trick pony, it works well on one table query.

SELECT COUNT(1) FROM tbl

But when you use joins, that trick won't work on multi table queries without its semantics being confused, and in particular you cannot write:

SELECT boss.boss_id, COUNT(subordinate.1)

FROM boss

LEFT JOIN subordinate ON subordinate.bos.id = boss.boss_id

GROUP BY boss_id

Summary

COUNT(*) Counts the number of rows

COUNT(1) Also counts the number of rows

11.2 Sum with Sum Function

Get the total domestic movie revenue

```
SELECT SUM(revenues_domestic) FROM movies_revenues;
```

	sum numeric
1	5719.50

Get the total domestic movie revenue where the movie revenue is greater than 200

```
SELECT SUM(revenues_domestic) FROM movies_revenues  
WHERE revenues_domestic > 200;
```

	sum numeric
1	3425.60

Find the total movie length of all English movies

```
SELECT SUM(movie_length) FROM movies  
WHERE movie_lang = 'English';
```

	sum bigint
1	4824

Can I sum all the movie names?

```
SELECT SUM(movie_name) FROM movies;  
ERROR: function sum(character varying) does not exist  
LINE 1: SELECT SUM(movie_name) FROM movies;  
                  ^
```

HINT: No function matches the given name and argument types. You might need to add explicit type casts.
SQL state: 42883 Character: 8

Sum is a numeric function and can only be used on numeric data types.

```
SELECT SUM(DISTINCT(revenues_domestic)) FROM movies_revenues;
```

Distinct will ignore null and duplicate values so if two movies have the same domestic revenue then only one will be counted.

	sum numeric
1	5708.40

11.3 Min & Max Functions

	max integer
1	168

```
SELECT MAX(movie_length) FROM movies;
```

Get the longest movie

Get the shortest movie

```
SELECT MIN(movie_length) FROM movies;
```

	min integer
1	87

```
SELECT MAX(movie_length) FROM movies  
WHERE movie_lang = 'English';
```

	max integer
1	168

Get the longest English language movie

```
SELECT MAX(movie_length) FROM movies  
WHERE movie_lang = 'English';
```

	max integer
1	168

Get the longest English language movie

```
SELECT MAX(release_date) FROM movies  
WHERE movie_lang = 'English';
```

	max date
1	2017-11-10

Get the release date of the latest released English language movie.

```
SELECT MIN(release_date) FROM movies  
WHERE movie_lang = 'Chinese';
```

	min date
1	1972-06-01

Get the first release date of a Chinese language movie.

```
SELECT MIN(movie_name) FROM movies;
```

	min text
1	A Clockwork Orange

Min and MAX work on text based data types. For example MIN would order the movies alphabetically then get the first. Max would get the last movie name alphabetically in the table

11.3 Greatest & Least Functions

The greatest and least functions will select the largest or smallest value from a list of expressions.

	greatest integer
1	30

```
SELECT GREATEST(10,20,30);
```

```
SELECT LEAST(10,-20,30);
```

	least integer
1	-20

Can we compare VARCHAR in a list? – YES

```
SELECT GREATEST('A','B','C');
```

	greatest text
1	C

Can we compare INT and VARCHAR in a list? – NO. They are not a common data type.

```
SELECT GREATEST(1,'A',2,3);  
ERROR: invalid input syntax for type integer: "A"  
LINE 1: SELECT GREATEST(1,'A',2,3);  
                        ^  
SQL state: 22P02 Character: 19
```

```
SELECT  
  movie_id,  
  revenues_domestic,  
  revenues_international,  
  GREATEST(revenues_domestic,revenues_international) AS GREATEST  
FROM movies_revenues;
```

	movie_id integer	revenues_domestic numeric (10,2)	revenues_international numeric (10,2)	greatest numeric (10,2)
1	45	22.20	1.30	22.20
2	13	199.40	201.20	201.20
3	23	102.10	[null]	102.10

Movie ID 45 made the most money domestically whereas movie 13 made the most money on international sales.

11.4 GREATEST() vs MAX()

At first glance, the MySQL MAX() and GREATEST() functions perform a similar operation. They both return the maximum value from a range of values. However, there is a difference between these two functions. The best way to see the difference is to compare the syntax.

```
MAX([DISTINCT] expr [over clause]);  
GREATEST(value1,value2,...);
```

So already they look completely different. The Max() function accepts the distinct keyword as well as an over clause and the greatest() function does not.

However, the key difference between these two functions is that MAX accepts only one argument and GREATEST accepts multiple arguments.

So MAX is typically used to return the maximum value in a column in a database. The table could contain many rows but this function returns the one with the maximum value.

GREATEST() on the other hand, returns the maximum-valued argument from the list of arguments passed to it. So you could pass, say 3 arguments to this function and it will return the one with the largest value.

```
SELECT MAX(population) AS result FROM city;  
SELECT GREATEST(1,5,9) AS result;
```

If you try to pass only one argument to GREATEST(), you will get an error.

11.5 Average with AVG function

Function to calculate the average on a set. Returns one value.

Syntax: **AVG(column_name)** or **SELECT AVG(Column_name) FROM table_name**

```
SELECT AVG(movie_length) FROM movies;
```

Get the average movie length for all movies. Note that AVG always returns a numeric value even though the movie length column is an integer.

	avg numeric	
1	126.1320754716981132	

```
SELECT AVG(movie_length) FROM movies  
WHERE movie_lang = 'English';
```

	avg numeric	
1	126.9473684210526316	

Get the average movie length for all English movies

```
SELECT AVG(movie_length) FROM movies  
WHERE movie_lang = 'Chinese';
```

	avg numeric	
1	121.8000000000000000	

Get the average movie length for all Chinese movies

```
SELECT AVG(DISTINCT(movie_length)) FROM movies  
WHERE movie_lang = 'English';
```

	avg numeric	
1	127.7407407407407407	

Distinct will ignore duplicates and nulls

```
SELECT AVG(movie_name) FROM movies;  
ERROR: function avg(character varying) does not exist  
LINE 1: SELECT AVG(movie_name) FROM movies;  
                  ^
```

HINT: No function matches the given name and argument types. You might need to add explicit type casts.

SQL state: 42883 Character: 8

AVG only works for numeric or integer data types

```
SELECT
    AVG(movie_length),
    SUM(movie_length)
FROM movies
WHERE movie_lang = 'English';
```

Get the total and average length of all English movies

	avg numeric	sum bigint
1	126.9473684210526316	4824

How does average handle NULL values?

```
CREATE TABLE demo_avg (
    num INT
);

INSERT INTO demo_avg (num)
VALUES
    (1),
    (2),
    (3),
    (NULL);

SELECT * FROM demo_avg;
```

	num integer
1	1
2	2
3	3
4	[null]

```
SELECT AVG(num) FROM demo_avg;
```

	avg numeric
1	2.0000000000000000

AVG will **ignore** null values

11.6 Combining Columns using Mathematical Operators

```
SELECT
2+10 AS addition,
10-2 AS subtraction,
11/2 AS division_integer,
11/2::numeric AS division_numeric,
2.56*10.23 AS multiplication_numeric,
2*10 AS multiplication_integer;
```

Note that the output of division by default is integer unless we cast it as numeric.

```
-- Arithmetic Operators
-----
-- addition          +
-- subtraction       -
-- multiplication    *
-- division          /
-- modulus           %
```

	addition integer	subtraction integer	division_integer integer	division_numeric numeric	multiplication_numeric numeric	multiplication_integer integer
1	12	8	5	5.5000000000000000	26.1888	20

```
SELECT
16%2 AS modulus_remainder,
15%6 AS modulus_remainder_2;
```

	modulus_remainder integer	modulus_remainder_2 integer
1	0	3

Get total movie revenue for all movies

```
SELECT
movie_id,
revenues_domestic,
revenues_international,
(revenues_international +
revenues_domestic) AS total_revenue
FROM movies_revenues;
```

	movie_id integer	revenues_domestic numeric (10,2)	revenues_international numeric (10,2)	total_revenue numeric
1	45	22.20	1.30	23.50
2	13	199.40	201.20	400.60
3	23	102.10	[null]	[null]
4	44	158.70	[null]	[null]

Note that arithmetic operators will ignore a null value so each time either value1 or value 2 is null then the result will be null

12. Using Date/Time Functions

[12.1 Datetimes Data Types](#)

[12.2 System Month Date Settings](#)

[12.3 Time of Day Input Formats](#)

[12.4 String to Dates Conversions](#)

[12.5 Using To Timestamp Function](#)

[12.6 Date Construction Functions](#)

[12.7 Make TimestampTZ Function](#)

[12.8 Date Value Extractor Functions](#)

[12.9 Math Operators with Dates](#)

[12.10 Overlaps Operator](#)

[12.11 Date/Time Functions](#)

[12.12 PostGreSQL Date/Time Functions](#)

[12.13 Age Function](#)

[12.14 Date Accuracy with Epoch](#)

[12.15 Using Date, Time & Timestamp in Tables](#)

[12.16 View & Set Timezone](#)

[12.17 How to Handle Timezones](#)

[12.18 Date Part function](#)

[12.19 Date Trunc function](#)

12.1 Datetimes Data Types

DATE	YYYY-MM-DD	2024-01-22	
TIME	HH:MM:SS	18:30:34	
TIME	HH:MM:SS.ppp	18:30:34.123	
TIME+TZ	HH:MM:SS+TZ	18:30:34+1	
TIME+TZ	HH:MM:SS.ppp+TZ	18:30:34.123+1	
TIMESTAMP	YYYY-MM-DD HH:MM:SS	2024-01-22	18:30:34
TIMESTAMP	YYYY-MM-DD HH:MM:SS.ppp	2024-01-22	18:30:34.123
TIMESTAMP+TZ	YYYY-MM-DD HH:MM:SS+TZ	2024-01-22	18:30:34+1
TIMESTAMP+TZ	YYYY-MM-DD HH:MM:SS.ppp+TZ	2024-01-22	18:30:34.123+1

YYYY = 4 digit year

MM = 2 digit Month

DD = 2 digit day

HH = 2 digit hours in 24h clock

MM = 2 digit minutes

SS = 2 digit seconds

ppp = precision in milliseconds

TZ = number of hours + or - of UTC

12.2 System Month Date Settings

SHOW DateStyle will display the current format of the date. THE default is ISO DMY I,e. Day Month Year

```
SHOW DateStyle;
```

	DateStyle text
1	ISO, DMY

```
SET DateStyle = 'ISO DMY';  
SET DateStyle = 'ISO MDY';  
SET DateStyle = 'ISO YMD';
```

DateStyle Can be set to a different format.

12.3 Time of Day Input Formats

HH:MM:SS

Internally Time is stored as HH:MM:SS as ISO 8601 format

HH:MM

Time can be displayed with AM and PM based on a 12 hour clock i.e. 10:30 AM

HH:MM:SS.123

Time can be displayed with extra precision in milliseconds up to six decimal places

YYYY-MM-DD HH:MM:SS

Timestamp is date and time together with optional precision and timezone

Allballs

represents the time

Yesterday

Date, Timestamp

NOW()

Date, Time, Timestamp

epoch

Date, Timestamp

Today

Date, Timestamp

infinity

Date, Timestamp

Tomorrow

Date, Timestamp

-infinity

Date, Timestamp

12.4 String to Dates Conversions

The To Date function will convert a string to a date. Syntax: `TO_DATE(date, format)`

Where the format can be any of the parameters defined on [slide 75](#).

```
SELECT TO_DATE('2020-01-01', 'YYYY-MM-DD');
```

	to_date date
1	2020-01-01

Note how the output is a date type when converting a string to a data

```
SELECT TO_DATE('20200101', 'YYYY-MM-DD');  
ERROR: date out of range: "20200101"  
SQL state: 22008
```

This has failed because PostgreSQL cannot understand it. In the string we have the date without hyphens and in the format we have the date specified with the hyphens.

```
SELECT TO_DATE('20200101', 'YYYYMMDD');
```

	to_date date
1	2020-01-01

Note how if I omit the hyphens in the format it is able to convert the string into a date type.

```
SELECT TO_DATE('01-01-2020', 'DD-MM-YYYY');
```

	to_date date
1	2020-01-01

Even if my input has Year and month backwards it will still convert it to a YYYY-MM-DD ISO date type.

```
SELECT TO_DATE('December 1, 2020', 'Month DD, YYYY');
```

	to_date	
	date	🔒
1	2020-12-01	

```
SELECT TO_DATE('Dec 1, 2020', 'Month DD, YYYY');
```

ERROR: The given value did not match any of the allowed values for this field.invalid value "Dec" for "Month"

ERROR: invalid value "Dec" for "Month"

SQL state: 22007 Detail: The given value did not match any of the allowed values for this field.

This has failed because PostgreSQL cannot understand it. In the string we have the month written as capital latter short form whereas we have specified capitalised long form in the format.

```
SELECT TO_DATE('Dec 1, 2020', 'Mon DD, YYYY');
```

	to_date	
	date	🔒
1	2020-12-01	

Now it works because 'Mon' is the correct format for short form month name with capitalised first letter.

```
SELECT TO_DATE('8th Dec, 2020', 'DDth Mon, YYYY');
```

	to_date	
	date	🔒
1	2020-12-08	

'th' format is used if the day has the 1st, 2nd, 3rd or xth.

12.5 Using To_Timestamp Function

Use to convert a sting to a timestamp. Syntax: `TO_Timestamp(date, format)`

Where the format can be any of the parameters defined on [slide 75](#).

```
SELECT TO_TIMESTAMP ('2020-01-01 10:30:20', 'YYYY-MM-DD HH:MI:SS');
```

Timestamp output contains timezone because it displays timestamp as time on the local machine and I am UTC+1

	to_timestamp timestamp with time zone 
1	2020-01-01 10:30:20+01

```
SELECT TO_TIMESTAMP ('2020-01-01 10:30:20', 'YYYY-MM-DD HH:MI');
```

This works because I am telling it only convert the HH and MI

	to_timestamp timestamp with time zone 
1	2020-01-01 10:30:00+01

```
SELECT TO_TIMESTAMP ('2020-01-01 10:30:20', 'YYYY-MM-DD HH');
```

This also works. Note how it is only converting the HH.

	to_timestamp timestamp with time zone 
1	2020-01-01 10:00:00+01

TO_TIMESTAMP validates the input string with minimal error checking

```
SELECT TO_TIMESTAMP ('2020-01-01 30:30:20', 'YYYY-MM-DD HH24:MI:SS');  
ERROR: date/time field value out of range: "2020-01-01 30:30:20"  
SQL state: 22008
```

It does not tell me which part of the input is out of range.

```
SELECT TO_TIMESTAMP ('2020-01-01 10:4', 'YYYY-MM-DD HH:MS');
```

10:4 is not hours and minutes but hours and milliseconds

	to_timestamp timestamp with time zone 
1	2020-01-01 10:00:00.4+01

```
SELECT
  TO_TIMESTAMP('01-01-2020 10:01', 'DD-MM-YYYY HH:MS'),
  TO_TIMESTAMP('01-01-2020 10:004', 'DD-MM-YYYY HH:MS');
```

	to_timestamp timestamp with time zone	to_timestamp timestamp with time zone
1	2020-01-01 10:00:00.01+01	2020-01-01 10:00:00.004+01

Miliseconds or microseconds after the decimal point. To get miliseconds I need three decimal places, To get microseconds I need 2 decimal places. PostgreSQL will take what is input after the decimal and convert it according to the level of precision.

12.6 Formatting Dates

PostgreSQL `TO_CHAR(expression, format)` function converts a timestamp, interval, integer, double precision or numeric value to a string. Valid numeric formats are on [slide 74](#) and valid timestamp formats are on [slide 75](#).

```
SELECT CURRENT_TIMESTAMP;
```

Is the current Date and Time with timezone

	current_timestamp timestamp with time zone
1	2025-01-22 00:17:31.788834+01

```
SELECT CURRENT_TIMESTAMP,
  TO_CHAR('2020-01-01 10:00:00'::TIMESTAMP, 'YYYY Month DD');
```

I cast the date time string as timestamp then select my format

	current_timestamp timestamp with time zone	to_char text
1	2025-01-22 00:19:25.981867+01	2020 January 01


```
SELECT CURRENT_TIMESTAMP,
TO_CHAR('2020-01-01 10:00:00'::TIMESTAMP, 'YYYY Month DD'),
TO_CHAR('2020-01-01 10:00:00'::TIMESTAMP, 'YYYY Month DD');
```

Note how current timestamp is to a six decimal precision. TO_CHAR converts to TEXT.

	current_timestamp timestamp with time zone	to_char text	to_char text
1	2025-01-22 00:22:19.269881+01	2020 January 01	2020 January 01

```
SELECT CURRENT_TIMESTAMP,
TO_CHAR('2020-01-01 10:00:00'::TIMESTAMP, 'YYYY Month DD'),
TO_CHAR('2020-01-01 10:00:00'::TIMESTAMP, 'YYYY Month DD'),
TO_CHAR('2020-01-01T10:45:55-6:00'::TIMESTAMP, 'YYYY Month DD hh:mm:ss tz');
```

	current_timestamp timestamp with time zone	to_char text	to_char text	to_char text
1	2025-01-22 01:15:34.224482+01	2020 January 01	2020 January 01	2020 January 01 05:01:55 cet

In the last example I am stating with the T that I want to include a timezone and I am setting it to -6 hours UTC

```
SELECT CURRENT_TIMESTAMP,
TO_CHAR('2020-01-01 10:00:00'::TIMESTAMP, 'YYYY Month DD'),
TO_CHAR('2020-01-01 10:00:00'::TIMESTAMP, 'YYYY Month DD'),
TO_CHAR('2020-01-01T10:45:55-6:00'::TIMESTAMP, 'YYYY Month DD hh:mm:ss tz'),
TO_CHAR('2020-01-01T10:45:55-6:00'::TIMESTAMP, 'FMonth DDth YYYY hh:mm:ss tz');
```

	current_timestamp timestamp with time zone	to_char text	to_char text	to_char text	to_char text
1	2025-01-22 01:19:52.849091+01	2020 January 01	2020 January 01	2020 January 01 05:01:55 cet	January 01st 2020 05:01:55 cet

In the last example I am using FM Month (Fill mode, suppress padding blanks & leading zero's)

```
SELECT
    movie_name,
    release_date,
    TO_CHAR(release_date, 'FMMonth FMDDth, YYYY')
FROM movies;
```

	movie_name character varying (100)	release_date date	to_char text
1	A Clockwork Orange	1972-02-02	February 2nd, 1972
2	Apocalypse Now	1979-08-15	August 15th, 1979
3	Battle Royale	2001-01-04	January 4th, 2001
4	Blade Runner	1982-06-25	June 25th, 1982
5	Chungking Express	1996-08-03	August 3rd, 1996

I can use FM on the DD format to remove the leading zeros for days 1 to 9 of the month

```
SELECT
    movie_name,
    release_date,
    TO_CHAR(release_date, 'FMMonth FMDDth, YYYY hh:mm:ss tz')
FROM movies;
```

	movie_name character varying (100)	release_date date	to_char text
1	A Clockwork Orange	1972-02-02	February 2nd, 1972 12:02:00 cet
2	Apocalypse Now	1979-08-15	August 15th, 1979 12:08:00 cest
3	Battle Royale	2001-01-04	January 4th, 2001 12:01:00 cet
4	Blade Runner	1982-06-25	June 25th, 1982 12:06:00 cest
5	Chungking Express	1996-08-03	August 3rd, 1996 12:08:00 cest

I can also add a timezone. Note how it changes from CET to CEST when summer time is in force.

12.6 Date Construction Functions

Syntax: `MAKE_DATE(YYYY,MM,DD)`

`MAKE_TIME(HH,MM.SS)`

`MAKE_TIMESTAMP(YYYY,MM,DD,HH,MM,SS)`

```
SELECT MAKE_DATE(2020,01,01);
```

	make_date date
1	2020-01-01

```
SELECT MAKE_TIME(2,3,4.05);
```

	make_time time without time zone
1	02:03:04.05

```
SELECT MAKE_TIME(2,14,4.05);
```

	make_time time without time zone
1	02:14:04.05

Note the data type is time without timezone

```
SELECT MAKE_TIMESTAMP(2020,01,01,10,30,45);
```

	make_timestamp timestamp without time zone
1	2020-01-01 10:30:45

12.7 Make_Interval Function

Syntax: MAKE_INTERVAL(Years, months, weeks, days, hours, minutes, seconds)

```
SELECT MAKE_INTERVAL(2020,01,01,01,10,30,45);
```

	make_interval interval	
1	2020 years 1 mon 8 days 10:30:45	

```
SELECT  
  MAKE_INTERVAL(2020,01,01,01,10,30,45),  
  MAKE_INTERVAL(2020,01,02,01,10,30,45),  
  MAKE_INTERVAL(2020,01,03,01,10,30,45),  
  MAKE_INTERVAL(2020,01,04,01,10,30,45),  
  MAKE_INTERVAL(2020,01,05,01,10,30,45),  
  MAKE_INTERVAL(2020,01,06,01,10,30,45);
```

Note how 1 week and 1 day is shown as 8 days. As I increase the week count, the days increase by 7.

make_interval interval	make_interval interval	make_interval interval
2020 years 1 mon 8 days 10:30:45	2020 years 1 mon 15 days 10:30:45	2020 years 1 mon 22 days 10:30:45
2020 years 1 mon 29 days 10:30:45	2020 years 1 mon 36 days 10:30:45	2020 years 1 mon 43 days 10:30:45

```
SELECT  
  MAKE_INTERVAL(days => 10),  
  MAKE_INTERVAL(months => 10, days => 2, mins => 35),  
  MAKE_INTERVAL(Weeks => 2);
```

	make_interval interval	make_interval interval	make_interval interval
1	10 days	10 mons 2 days 00:35:00	14 days

I can also used named notation to specify the interval.

12.7 Make_TimestampTZ Function

Syntax: MAKE_TIMESTAMPZ(Years, months, days, hours, minutes, seconds.ppp, tz)

```
SELECT MAKE_TIMESTAMPZ(2020,02,15,10,35,15.35);
```

	make_timestampz timestamp with time zone
1	2020-02-15 10:35:15.35+01

Note in the above example I did not specify the timezone so it defaulted to the local time on my SQL server which is running on my local machine.

```
SELECT pg_typeof(MAKE_TIMESTAMPZ(2020,02,15,10,35,15.35));
```

	pg_typeof regtype
1	timestamp with time zone

The above utility will tell me the data type.

```
SELECT * FROM pg_timezone_names;
```

	name text	abbrev text	utc_offset interval	is_dst boolean
453	Europe/Ljubljana	CET	01:00:00	false
454	Europe/London	GMT	00:00:00	false
455	Europe/Luxembourg	CET	01:00:00	false
456	Europe/Madrid	CET	01:00:00	false
457	Europe/Malta	CET	01:00:00	false

```
SELECT * FROM pg_timezone_abbrevs;
```

	abbrev text	utc_offset interval	is_dst boolean
40	CCT	08:00:00	false
41	CDT	-05:00:00	true
42	CEST	02:00:00	true
43	CET	01:00:00	false
44	CETDST	02:00:00	true

I can also see the full list of timezones and timezone abbreviations

The columns are: Abbrev = abbreviation, UTC_offset is the plus or minus hours from UTC, is_dst = Boolean for is daylight saving time.

```
SELECT pg_typeof(MAKE_TIMESTAMPTZ(2020,02,15,10,35,15.35, 'Asia/Hong_Kong'));
```

I can verify if the name of a timezone is valid.

pg_typeof regtype	
1	timestamp with time zone

```
SELECT MAKE_TIMESTAMPTZ(2020,02,15,00,00,00.000, 'Asia/Hong_Kong');
```

I can get a timestamptz for a particular date, time and timezone. Note hong-kong is 8 hours ahead of UTC.

make_timestamptz timestamp with time zone	
1	2020-02-14 17:00:00+01

```
SELECT
MAKE_TIMESTAMPTZ(2020,01,01,01,01,00.00, 'EST'),
MAKE_TIMESTAMPTZ(2020,01,01,01,01,00.00, 'PST'),
MAKE_TIMESTAMPTZ(2020,01,01,01,01,00.00, 'ACDT'),
MAKE_TIMESTAMPTZ(2020,01,01,01,01,00.00, 'ACT'),
MAKE_TIMESTAMPTZ(2020,01,01,01,01,00.00, '+1'),
MAKE_TIMESTAMPTZ(2020,01,01,01,01,00.00, '-1'),
MAKE_TIMESTAMPTZ(2020,01,01,01,01,00.00, 'NZ');
```

The date and time is automatically adjusted for the corresponding timezone then the output is given relative to the timezone set on the server which in this case is on my local machine and running in UTC+1

make_timestamptz timestamp with time zone	make_timestamptz timestamp with time zone	make_timestamptz timestamp with time zone	make_timestamptz timestamp with time zone
2020-01-01 07:01:00+01	2020-01-01 10:01:00+01	2019-12-31 15:31:00+01	2020-01-01 07:01:00+01

make_timestamptz timestamp with time zone	make_timestamptz timestamp with time zone	make_timestamptz timestamp with time zone
2020-01-01 01:01:00+01	2020-01-01 03:01:00+01	2019-12-31 13:01:00+01

12.8 Date Value Extractor Functions

Quick Syntax to extract data from date value:

EXTRACT(field FROM source)

DATE_PART(field, source)

The Date components on [slide 164](#) can be extracted

```
SELECT  
EXTRACT('DAY' FROM CURRENT_TIMESTAMP) AS "Day",  
EXTRACT('MONTH' FROM CURRENT_TIMESTAMP) AS "Month",  
EXTRACT('YEAR' FROM CURRENT_TIMESTAMP) AS "Year",  
EXTRACT('EPOCH' FROM CURRENT_TIMESTAMP) AS "Epoch";
```

	Day numeric	Month numeric	Year numeric	Epoch numeric
1	22	1	2025	1737586709.849149

```
SELECT EXTRACT('CENTURY' FROM INTERVAL '500 YEARS 2 MONTHS 11 DAYS');
```

An interval can be extracted from date and time components

	extract numeric
1	5

12.9 Math Operators with Dates

```
SELECT DATE '20200101' + 10;
```

Here the maths is adding 10 to the string then casting it as a date

	?column? date
1	2020-01-11

I can explicitly cast a string as a date before doing the addition

```
SELECT DATE '20200101'::DATE + 10;
```

	?column? date
1	2020-01-11

```
SELECT
TIME '23:59:59' + INTERVAL '1 SECOND',
TIME '23:59:59' + INTERVAL '2 SECOND';
```

	?column? time without time zone	?column? time without time zone
1	00:00:00	00:00:01

The interval can be used on time or date

```
SELECT
TIME '23:59:59' + INTERVAL '1 SECOND',
TIME '23:59:59' + INTERVAL '2 SECOND';
```

	?column? time without time zone	?column? time without time zone
1	00:00:00	00:00:01

The interval can be used on time or date

```
SELECT
CURRENT_TIMESTAMP,
CURRENT_TIMESTAMP + '01:01:01';
```

	current_timestamp timestamp with time zone	?column? timestamp with time zone
1	2025-01-23 00:39:41.020719+01	2025-01-23 01:40:42.020719+01

I can add time to a timestamp

```
SELECT
DATE '20200101' + TIME '10:25:10',
SELECT '10:10:10' + TIME '10:25:10';
```

	?column? timestamp without time zone	?column? time without time zone
1	20200-10-10 10:25:10	20:35:20

I can add time to a date. In the second example because I have not specified TIME it is assuming that it is time and just returning time

```
SELECT
DATE '20200101' - INTERVAL '1 HOUR' AS "An Hour Ago",
INTERVAL '30 MINUTES' + INTERVAL '30 MINUTES';
```

	An Hour Ago timestamp without time zone	?column? interval
1	2019-12-31 23:00:00	01:00:00

I can add or subtract intervals

```
SELECT
INTERVAL '30 MINUTES' + INTERVAL '15 MINUTES' AS "quarter hour",
INTERVAL '30 MINUTES' + INTERVAL '1 DAY' AS "1800 seconds",
INTERVAL '2:00' / 2 AS "1 hour";
```

	quarter hour interval	1800 seconds interval	1 hour interval
1	00:45:00	1 day 00:30:00	01:00:00

Following fields can be used		
Field Value	TIMESTAMP	Interval
CENTURY	The century	The number of Centuries
DAY	Day of month (1-31)	The number of days
DECADE	The decade that is the year divided by 10	Same as TIMESTAMP
DOW	Day of Week from Sunday (0) to Saturday (6)	N/A
DOY	Day of year (1-366)	N/A
EPOCH	Number of seconds since 1-Jan-1970 00:00:00 UTC	Total number of seconds in the interval
HOUR	Hour (0-23)	
ISODOW	Day of Week based on ISO 8601 Monday (1) to Sunday (7)	N/A
ISOYEAR	ISO 8601 week number of year	N/A
MICROSECONDS	Seconds field including decimal precision part	Same as TIMESTAMP
MILLENNIUM	The millennium	The number of millennium
MILISECONDS	The second field, including decimal precision part	Same as TIMESTAMP
MINUTE	Minute (0-59)	The number of minutes
MONTH	Month (1-12)	The number of months, modulo (0-11)
QUARTER	The quarter of the year	The number of quarters
SECOND	Second	The number of seconds
TIMEZONE	The timezone offset from UTC, measured in seconds	N/A
TIMEZONE_HOUR	The hour component of the timezone offset	N/A
TIMEZONE_MINUTE	The minute component of the timezone offset	N/A
WEEK	The ISO 8601 week number	N/A
YEAR	The year	Same as TIMESTAMP

12.10 Overlaps Operator

```
SELECT (DATE '2020-01-01', DATE '2020-12-31') OVERLAPS (DATE '2020-10-12', DATE '2020-12-01');
```

	overlaps boolean 
1	true

Overlap is used to compare a date range specified as a start date and an end date to check if it overlaps with another date range specified as a start date and an end date.

```
SELECT (DATE '2020-01-01', INTERVAL '100 DAYS') OVERLAPS (DATE '2020-10-12', DATE '2020-12-01');
```

	overlaps boolean 
1	false

I can also specify a start date and an interval from that date to compare with another date range.

```
SELECT (DATE '2020-01-01', DATE '2020-11-30') OVERLAPS (DATE '2020-12-12', DATE '2020-12-01');
```

	overlaps boolean 
1	false

Note how the start date is after the end date on the right side overlapping argument but the query still works.

	overlaps boolean 
1	true

```
SELECT (DATE '2020-01-01', DATE '2020-12-31') OVERLAPS (DATE '2020-12-12', DATE '2020-12-01');
```

12.11 Date/Time Functions

```
SELECT
CURRENT_DATE,
CURRENT_TIME,
CURRENT_TIMESTAMP,
LOCALTIME,
LOCALTIMESTAMP;
```

Current date, Current time and current timestamp are the server times. Local time and local timestamp are the times converted to the TZ where the query is being run from. In this case my server is on my local machine so local time is the same as server time.

	current_date date	current_time time with time zone	current_timestamp timestamp with time zone	localtime time without time zone	localtimestamp timestamp without time zone
1	2025-01-23	01:32:06.827907+01:00	2025-01-23 01:32:06.827907+01	01:32:06.827907	2025-01-23 01:32:06.827907

```
SELECT
CURRENT_DATE,
CURRENT_TIME(1),
CURRENT_TIMESTAMP(2),
LOCALTIME(3),
LOCALTIMESTAMP(4);
```

I can use precision with time specifying the decimal value of seconds up to six places. (1 to 6)

	current_date date	current_time time with time zone (1)	current_timestamp timestamp with time zone (2)	localtime time without time zone (3)	localtimestamp timestamp without time zone (4)
1	2025-01-23	01:36:07.800000+01:00	2025-01-23 01:36:07.78+01	01:36:07.782	2025-01-23 01:36:07.7822

```
CREATE table logs(
    log_id SERIAL PRIMARY KEY,
    add_date DATE DEFAULT CURRENT_DATE,
    add_time TIME DEFAULT CURRENT_TIME
);

INSERT INTO logs (log_id) VALUES (1);
```

	log_id [PK] integer	add_date date	add_time time without time zone
1	1	2025-01-23	02:31:28.852667

Date/time functions can be used as default values for columns.

12.12 PostGreSQL Date/Time Functions

```
SELECT  
NOW(),  
TRANSACTION_TIMESTAMP(),  
STATEMENT_TIMESTAMP(),  
CLOCK_TIMESTAMP();
```

Transaction timestamp is the same as now.

Statement_timestamp is the time execution of this statement.

Clock_timestamp shows the current date and time (changes during statement execution)

	now timestamp with time zone	transaction_timestamp timestamp with time zone	statement_timestamp timestamp with time zone	clock_timestamp timestamp with time zone
1	2025-01-23 01:43:51.485345+01	2025-01-23 01:43:51.485345+01	2025-01-23 01:43:51.485345+01	2025-01-23 01:43:51.486317+01

Notice that Now, Transaction and statement all show the same time but the clock is a thousands of milliseconds later after the query has executed.

```
SELECT TIMEOFDAY();
```

	timeofday text
1	Thu Jan 23 01:51:11.884287 2025 CET

Outputs a human readable exact timestamp of current time to precision of six decimal places

12.13 Age Function

Returns the number of years, Months and Days between two dates. Syntax: **AGE(Date1, Date2)**

```
SELECT AGE('2021-01-01', '2019-01-01');
```

	age interval 
1	2 years

```
SELECT AGE('2019-01-01');  
ERROR: function age(unknown) is not unique
```

```
LINE 1: SELECT AGE('2019-01-01');  
                ^
```

HINT: Could not choose a best candidate function. You might need to add explicit type casts.

SQL state: 42725 Character: 8

	age interval 
1	6 years 22 days

```
SELECT AGE(TIMESTAMP '2019-01-01');
```

```
SELECT AGE(CURRENT_DATE, TIMESTAMP '2019-01-01');
```

If I typecast the date but only include one date then it will subtract the current date from the specified date.

```
SELECT AGE('2010-01-01', '2020-10-01');
```

If date1 is before date2 then it will calculate as a negative number

	age interval 
1	-10 years -9 mons

12.14 Date Accuracy with Epoch

Arithmetic with dates are sometimes complex, and for more accuracy consider Timezones and Daylight saving times. Age function can help in the above, but it may not yield an accurate result so for that we use EPOCH.

```
SELECT AGE(timestamp '2020-12-20', TIMESTAMP '2020-10-20');
```

	age interval	
1	2 mons	

This yields the months only and not seconds

```
SELECT  
EXTRACT(EPOCH FROM TIMESTAMPTZ '2020-12-20') -  
EXTRACT(EPOCH FROM TIMESTAMPTZ '2020-10-20')  
AS "difference in seconds";
```

	difference in seconds numeric	
1	5274000.000000	

Epoch converts datetime into a number as seconds from 1st jan 1970 so this is accurate to six decimal places of a second.

```
SELECT  
(EXTRACT(EPOCH FROM TIMESTAMPTZ '2020-12-20') -  
EXTRACT(EPOCH FROM TIMESTAMPTZ '2020-10-20'))  
/ 60 / 60 / 24 AS "difference in hours";
```

	difference in hours numeric	
1	61.041666666666667	

```
SELECT  
(EXTRACT(EPOCH FROM TIMESTAMPTZ '2020-12-20') -  
EXTRACT(EPOCH FROM TIMESTAMPTZ '2020-10-20'))  
/ 60 / 60 / 24 AS "difference in hours",  
TIMESTAMP '2020-12-20 12:00:00' -  
TIMESTAMP '2020-10-20 12:00:00'  
AS "timestamptz",  
AGE(TIMESTAMPTZ '2020-12-20 12:00:00',  
TIMESTAMPTZ '2020-10-20 12:00:00')  
AS "age";
```

	difference in hours numeric	timestamptz interval	age interval
1	61.041666666666667	61 days	2 mons

Age is the least accurate just subtracting the difference between two dates. TimestampTZ is more accurate but EPOCH is the most precise.

12.15 Using Date, Time & Timetamp in Tables

```
CREATE table times (  
  time_id SERIAL PRIMARY KEY,  
  start_date DATE,  
  start_time TIME,  
  start_timestamp TIMESTAMP  
);
```

Below is a list of common date and time words. The format specifies the data types that they are compatible with. We can use list table to insert them into the table

```
INSERT INTO times (  
  start_date,  
  start_time,  
  start_timestamp)  
VALUES (  
  'epoch',  
  'allballs',  
  'infinity'  
);  
  
SELECT * FROM times;
```

Name	Format	Notes
Allballs	time	00:00:00 UTC
NOW()	Date, Time, Timestamp	current transaction datetime
Today	Date, Timestamp	Midnight today
Tomorrow	Date, Timestamp	Midnight tomorrow
Yesterday	Date, Timestamp	Midnight yesterday
epoch	Date, Timestamp	1970-01-01 00:00:00 UTC
infinity	Date, Timestamp	Latter than all other timestamps
-infinity	Date, Timestamp	Earlier than all other timestamps

	time_id [PK] integer	start_date date	start_time time without time zone	start_timestamp timestamp without time zone
1	1	1970-01-01	00:00:00	infinity

	time_id [PK] integer	start_date date	start_time time without time zone	start_timestamp timestamp without time zone
1	1	1970-01-01	00:00:00	infinity
2	2	1970-01-01	00:00:00	-infinity

```
INSERT INTO times (  
  start_date,  
  start_time,  
  start_timestamp)  
VALUES (  
  'epoch',  
  'allballs',  
  '-infinity'  
);  
  
SELECT * FROM times;
```

```
INSERT INTO times (
  start_date,
  start_time,
  start_timestamp)
VALUES (
  now(),
  'allballs',
  '-infinity'
);

SELECT * FROM times;
```

	time_id [PK] integer	start_date date	start_time time without time zone	start_timestamp timestamp without time zone
1	1	1970-01-01	00:00:00	infinity
2	2	1970-01-01	00:00:00	-infinity
3	3	2025-01-23	00:00:00	-infinity

12.16 View & Set Timezone

```
SELECT * FROM pg_timezone_names;
```

	name text	abbrev text	utc_offset interval	is_dst boolean
453	Europe/Ljubljana	CET	01:00:00	false
454	Europe/London	GMT	00:00:00	false
455	Europe/Luxembourg	CET	01:00:00	false
456	Europe/Madrid	CET	01:00:00	false

```
SELECT * FROM pg_timezone_abbrevs;
```

	abbrev text	utc_offset interval	is_dst boolean
40	CCT	08:00:00	false
41	CDT	-05:00:00	true
42	CEST	02:00:00	true
43	CET	01:00:00	false

I can also see the full list of timezones and timezone abbreviations

The columns are: Abbrev = abbreviation, UTC_offset is the plus or minus hours from UTC, is_dst = Boolean for is daylight saving time. <https://www.postgresql.org/docs/7.2/timezones.html>

```
SHOW TIME_ZONE;
```

	TimeZone text
1	Europe/Paris

```
SET TIME_ZONE 'UTC';
SET Query returned successfully in 50 msec.
```

I can change my server's default timezone.

```
SET TIME_ZONE 'Europe/Madrid';
```

```
SHOW TIME_ZONE;
```

	TimeZone text
1	UTC

12.17 How to Handle Timezones

```
ALTER TABLE times
ADD COLUMN end_timestamp TIMESTAMP WITH TIME ZONE;

ALTER TABLE times
ADD COLUMN end_time TIME WITH TIME ZONE;

INSERT INTO times(end_timestamp, end_time) VALUES
('2020-06-20 11:30:00 US/Pacific', '11:30:00+6');

SELECT * FROM times;
```

In a multi-user, multi-location database application, timezones can quite complex and tricky and need to consider timezones and daylight savings time so it is always better to use date and time for global databases. Try avoid using numeric offsets with timezones.

	time_id [PK] integer	start_date date	start_time time without time zone	start_timestamp timestamp without time zone	end_timestamp timestamp with time zone	end_time time with time zone
1	1	1970-01-01	00:00:00	infinity	[null]	[null]
2	2	1970-01-01	00:00:00	-infinity	[null]	[null]
3	3	2025-01-23	00:00:00	-infinity	[null]	[null]
4	5	[null]	[null]	[null]	2020-01-20 20:30:00+01	11:30:00+06:00

Note how in **end_timestamp** it has automatically converted the US/Pacific timezone to the correct time for the server which is UTC+1. note how for **end_time** it has added the time with the UTC offset

```
INSERT INTO times(end_timestamp, end_time) VALUES
('2020-06-20 11:30:00 US/Pacific', '11:30:00+6');

SELECT * FROM times;
```

Note how postgreSQL takes into account that in the US/Pacific in June it is daylight savings hours so the timezone is +2

	time_id [PK] integer	start_date date	start_time time without time zone	start_timestamp timestamp without time zone	end_timestamp timestamp with time zone	end_time time with time zone
1	1	1970-01-01	00:00:00	infinity	[null]	[null]
2	2	1970-01-01	00:00:00	-infinity	[null]	[null]
3	3	2025-01-23	00:00:00	-infinity	[null]	[null]
4	5	[null]	[null]	[null]	2020-01-20 20:30:00+01	11:30:00+06:00
5	6	[null]	[null]	[null]	2020-06-20 20:30:00+02	11:30:00+06:00

```
SET TIME ZONE 'UTC';
SELECT * FROM times;
```

If I change the timezone then postgreSQL will automatically adjust the dateTimeTZ outputs.

	time_id [PK] integer	start_date date	start_time time without time zone	start_timestamp timestamp without time zone	end_timestamp timestamp with time zone	end_time time with time zone
1	1	1970-01-01	00:00:00	infinity	[null]	[null]
2	2	1970-01-01	00:00:00	-infinity	[null]	[null]
3	3	2025-01-23	00:00:00	-infinity	[null]	[null]
4	5	[null]	[null]	[null]	2020-01-20 19:30:00+00	11:30:00+06:00
5	6	[null]	[null]	[null]	2020-06-20 18:30:00+00	11:30:00+06:00

Note that this does not change the data in the table, the dateTimeTZ is absolute. It changes the timestamp based on the location where the query is being run from.

```
SET TIME ZONE 'US/Pacific';
SELECT * FROM times;
```

	time_id [PK] integer	start_date date	start_time time without time zone	start_timestamp timestamp without time zone	end_timestamp timestamp with time zone	end_time time with time zone
1	1	1970-01-01	00:00:00	infinity	[null]	[null]
2	2	1970-01-01	00:00:00	-infinity	[null]	[null]
3	3	2025-01-23	00:00:00	-infinity	[null]	[null]
4	5	[null]	[null]	[null]	2020-01-20 11:30:00-08	11:30:00+06:00
5	6	[null]	[null]	[null]	2020-06-20 11:30:00-07	11:30:00+06:00

```
SET TIME ZONE 'Europe/Madrid';
```

	time_id [PK] integer	start_date date	start_time time without time zone	start_timestamp timestamp without time zone	end_timestamp timestamp with time zone	end_time time with time zone
1	1	1970-01-01	00:00:00	infinity	[null]	[null]
2	2	1970-01-01	00:00:00	-infinity	[null]	[null]
3	3	2025-01-23	00:00:00	-infinity	[null]	[null]
4	5	[null]	[null]	[null]	2020-01-20 20:30:00+01	11:30:00+06:00
5	6	[null]	[null]	[null]	2020-06-20 20:30:00+02	11:30:00+06:00

12.18 Date_Part function

PostgreSQL DATE_PART() function allows the extraction of subfields from a date or time value.

The following fields can be extracted: Century, Decade, year, quarter, month, day, hour, minute, seconds, milliseconds, microsecond, dow, doy, isodow, isoyear, timezone, timezone_hour, timezone_minute.

Syntax: DATE_PART(field, source);

```
SELECT DATE_PART('year', TIMESTAMP '2017-01-01');
```

	date_part double precision
1	2017

```
SELECT
DATE_PART('year', TIMESTAMP '2017-01-01') AS "year",
DATE_PART('quarter', TIMESTAMP '2017-01-01') AS "quarter",
DATE_PART('month', TIMESTAMP '2017-01-01') AS "month",
DATE_PART('decade', TIMESTAMP '2017-01-01') AS "decade",
DATE_PART('century', TIMESTAMP '2017-01-01') AS "century";
```

	year double precision	quarter double precision	month double precision	decade double precision	century double precision
1	2017	1	1	201	21

```
SELECT
DATE_PART('week', TIMESTAMP '2017-01-01') AS "week",
DATE_PART('dow', TIMESTAMP '2017-01-01') AS "dow",
DATE_PART('doy', TIMESTAMP '2017-01-01') AS "doy",
DATE_PART('day', TIMESTAMP '2017-01-01') AS "day",
DATE_PART('hour', TIMESTAMP '2017-01-01 10:20:30') AS "hour",
DATE_PART('minute', TIMESTAMP '2017-01-01 10:20:30') AS "minute",
DATE_PART('second', TIMESTAMP '2017-01-01 10:20:30') AS "second";
```

	week double precision	dow double precision	doy double precision	day double precision	hour double precision	minute double precision	second double precision
1	52	0	1	1	10	20	30

Note that the 1st of January 2017 occurred in the 52nd week of 2016

```
SELECT
movie_name,
release_date,
DATE_PART('week', release_date) AS "release week",
DATE_PART('month', release_date) AS "release Month"
FROM movies;
```

	movie_name character varying (100)	release_date date	release week double precision	release Month double precision
1	A Clockwork Orange	1972-02-02	5	2
2	Apocalypse Now	1979-08-15	33	8
3	Battle Royale	2001-01-04	1	1
4	Blade Runner	1982-06-25	25	6

```
SELECT
movie_name,
release_date,
DATE_PART('week', release_date) AS "release week",
DATE_PART('month', release_date) AS "release Month"
FROM movies
ORDER BY 4 asc;
```

	movie_name character varying (100)	release_date date	release week double precision	release Month double precision
45	Leon	1994-11-18	46	11
46	Three Billboards Outside Ebbing, Misso...	2017-11-10	45	11
47	Toy Story	1995-11-22	47	11
48	Rushmore	1998-11-12	46	11
49	Raging Bull	1980-11-14	46	11
50	Titanic	1997-12-19	51	12
51	There Will Be Blood	2007-12-26	52	12

Here I can get an overview of when in the year the movies were released and see that some of the biggest movies were released later in the year.

12.19 Date_Trunc function

PostgreSQL DATE_TRUNC() function will truncate a timestamp or interval up to a specific level of precision based on a specific date part. E.g hour, week or month.

The following dateparts can be truncated: Millennium, Century, Decade, year, quarter, month, day, hour, minute, seconds, milliseconds, microsecond. The field can be a timestamp or interval.

Syntax: **DATE_TRUNC(datepart, field);**

```
SELECT DATE_TRUNC('hour', TIMESTAMP '2020-10-01 05:15:45');
```

```
SELECT
DATE_TRUNC('hour', TIMESTAMP '2020-10-01 05:15:45') AS "hour",
DATE_TRUNC('minute', TIMESTAMP '2020-10-01 05:15:45') AS "minute",
DATE_TRUNC('second', TIMESTAMP '2020-10-01 05:15:45') AS "second";
```

	hour timestamp without time zone	minute timestamp without time zone	second timestamp without time zone
1	2020-10-01 05:00:00	2020-10-01 05:15:00	2020-10-01 05:15:45

	date_trunc timestamp without time zone
1	2020-10-01 05:00:00

```
SELECT
DATE_PART('month', release_date) "release_month",
COUNT (movie_id)
FROM movies
GROUP BY release_month
ORDER BY 2 DESC;
```

Count the number of movies by release_month

	release_month double precision	count bigint
1	5	9
2	8	6
3	6	6
4	11	6
5	2	5
6	3	5
7	9	4
8	7	4
9	12	4
10	10	2
11	1	2

13. Grouping Data

[13.1 Group By](#)

[13.2 Use Group By with Multiple Columns, Order By](#)

[13.3 Order of Execution in Group By Clause](#)

[13.4 Using Having Clause](#)

[13.5 Having Vs Where](#)

[13.6 Handling Null Values with Group By using Coalesce](#)

13.1 Group By

The GROUP BY clause divides the rows returned from the select statement into groups. For each group an aggregate function can be applied like count, sum, min, max etc. Group by eliminates duplicate values from the results similar to DISTINCT. Syntax:

SELECT

Column 1

AGGREGATE_FUNCTION(column2)

FROM table_name

GROUP BY column1;

```
SELECT
SUM (movie_length),
age_certificate
FROM movies
GROUP BY age_certificate
ORDER BY 2 DESC;
```

Get sum total movie
length per age certificate

	sum bigint	age_certificate character varying (10)
1	620	U
2	1462	PG
3	994	18
4	2184	15
5	1425	12

```
SELECT
movie_lang,
COUNT (movie_lang)
FROM movies
GROUP BY movie_lang
ORDER BY 2 DESC;
```

Get average movie
length grouped by
movie language

	avg numeric	count bigint
1	126.9473684210526316	38
2	121.8000000000000000	5
3	111.5000000000000000	4
4	142.5000000000000000	2
5	128.0000000000000000	1
6	98.0000000000000000	1
7	130.0000000000000000	1
8	165.0000000000000000	1

```
SELECT
movie_lang,
COUNT (movie_lang)
FROM movies
GROUP BY movie_lang
ORDER BY 2 DESC;
```

Get the total count
of all movies by
movie language

	movie_lang character varying (20)	count bigint
1	English	38
2	Chinese	5
3	Japanese	4
4	Portuguese	2
5	Swedish	1
6	Spanish	1
7	Korean	1
8	German	1


```
SELECT
MIN(movie_length),
MAX(movie_length),
movie_lang
FROM movies
GROUP BY movie_lang
ORDER BY 3 DESC;
```

List min and max
movie lengths
grouped by movie
language

	min integer	max integer	movie_lang character varying (20)
1	128	128	Swedish
2	98	98	Spanish
3	140	145	Portuguese
4	130	130	Korean
5	107	120	Japanese
6	165	165	German
7	87	168	English
8	99	139	Chinese

```
SELECT
movie_length,
FROM movies
GROUP BY
movie_length;
```

Can we use group
by without an
aggregate
function? - YES

	movie_length integer
1	99
2	87
3	117
4	121
5	138
6	119
7	113
8	132

It will remove duplicates and take
only unique rows

13.2 Use Group By with Multiple Columns, Order By

```
SELECT
MIN(movie_length),
MAX(movie_length),
movie_lang
FROM movies
ORDER BY 3 DESC;
ERROR: column "movies.movie_lang" must appear in the GROUP BY clause or be used in an aggregate function
LINE 4: movie_lang
        ^
```

Can we use a column1 aggregate function without specifying group by clause? - **NO**

SQL state: 42803 Character: 46

```
SELECT
movie_lang,
age_certificate,
AVG(movie_length),
FROM movies
GROUP BY movie_lang
ORDER BY 3 DESC;
ERROR: column "movies.age_certificate" must appear in the GROUP BY clause or be used in an aggregate function
LINE 3: age_certificate,
        ^
```

Can we use more than 1 column in the select? without specifying group by clause for the second column? - **NO**

SQL state: 42803 Character: 20

```
SELECT
movie_lang,
age_certificate,
AVG(movie_length) AS "avg Movie length"
FROM movies
GROUP BY movie_lang, age_certificate
ORDER BY 1 DESC;
```

Get average movie length grouped by movie language and age certificate

	movie_lang character varying (20)	age_certificate character varying (10)	avg Movie length numeric
1	Swedish	15	128.0000000000000000
2	Spanish	PG	98.0000000000000000
3	Portuguese	15	140.0000000000000000
4	Portuguese	18	145.0000000000000000
5	Korean	18	130.0000000000000000
6	Japanese	18	109.5000000000000000
7	Japanese	U	113.5000000000000000

```
SELECT
movie_lang,
age_certificate,
AVG(movie_length) AS "avg Movie length"
FROM movies
WHERE movie_length > 100
GROUP BY movie_lang, age_certificate
ORDER BY 3 DESC;
```

Get average movie length grouped by movie language and age certificate where movie length is greater than 100

	movie_lang character varying (20)	age_certificate character varying (10)	avg Movie length numeric
1	German	15	165.0000000000000000
2	Portuguese	18	145.0000000000000000
3	Portuguese	15	140.0000000000000000
4	English	15	136.5000000000000000
5	English	12	132.7142857142857143
6	Chinese	12	132.3333333333333333
7	Korean	18	130.0000000000000000
8	Swedish	15	128.0000000000000000

Note that the where clause has to come before the group by and order by clause.

```
SELECT
age_certificate,
AVG(movie_length) AS "avg Movie length"
FROM movies
WHERE age_certificate = 'PG'
GROUP BY age_certificate
ORDER BY 1 DESC;
```

	age_certificate character varying (10) 🔒	avg Movie length numeric 🔒
1	PG	121.83333333333333

Get average movie length grouped by age certificate where age certificate is PG

```
SELECT
nationality,
COUNT(*)
FROM directors
GROUP BY nationality;
```

How many directors are there for each nationality?

Get total sum movie length for each age certificate and movie language combination

```
SELECT
movie_lang,
age_certificate,
SUM(movie_length)
FROM movies
GROUP BY movie_lang, age_certificate
ORDER BY 1 DESC;
```

	movie_lang character varying (20) 🔒	age_certificate character varying (10) 🔒	sum bigint 🔒
1	Swedish	15	128
2	Spanish	PG	98
3	Portuguese	15	140
4	Portuguese	18	145

```
SELECT
movie_lang,
age_certificate,
SUM(movie_length)
FROM movies
GROUP BY SUM(movie_length)
ORDER BY 1 DESC;
```

Can we use an aggregate function to group by? - **NO**

```
ERROR: aggregate functions are not allowed in GROUP BY
LINE 6: GROUP BY SUM(movie_length)
                  ^
SQL state: 42803 Character: 76
```

13.3 Order of Execution in Group By Clause

FROM	- Specify table
WHERE	- Conditions
GROUP BY	- Group sets
HAVING	- Filter again
SELECT	- Columns
DISTINCT	- Unique columns
ORDER BY	- Sort
LIMIT	- filter records

PostgreSQL evaluates the GROUP BY clause after the FROM and WHERE clauses but before the HAVING, SELECT, DISTINCT, ORDER BY and LIMIT clauses.

13.4 Using Having Clause

We use HAVING clause to specify a search condition for a group or aggregate

The HAVING clause is often used with the GROUP BY clause to filter rows based on filter conditions

Can not use column aliases with HAVING clause because it is evaluated before SELECT clause

HAVING clause calculates on the aggregate function and not actual select columns

Syntax:

```
SELECT  
Column1,  
AGGREGATE_FUNCTION(column2)  
FROM table_name  
GROUP BY column1  
HAVING condition;
```

```
SELECT
movie_lang,
SUM(movie_length)
FROM movies
GROUP BY movie_lang
HAVING SUM(movie_length) > 200
ORDER BY SUM(movie_length);
```

	movie_lang character varying (20) 🔒	sum bigint 🔒
1	Portuguese	285
2	Japanese	446
3	Chinese	609
4	English	4824

List movies language where sum total length of the movies is greater than 200

```
SELECT
director_id,
SUM(movie_length) AS "tml"
FROM movies
GROUP BY director_id
HAVING tml > 200
ORDER BY 2;
ERROR: column "tml" does not exist
LINE 6: HAVING tml > 200
                  ^
SQL state: 42703 Character: 89
```

Can I use an alias in a having clause? - **NO**

```
SELECT
director_id,
SUM(movie_length)
FROM movies
GROUP BY director_id
HAVING SUM(movie_length) > 200
ORDER BY SUM(director_id);
```

List directors where their sum total movie length is greater than 200.

	director_id integer 🔒	sum bigint 🔒
1	3	340
2	5	272
3	8	243
4	15	268
5	13	368
6	21	227
7	17	412
8	27	286
9	35	237
10	24	345
11	26	397

Having clause is evaluated before the select clause so when I try and use an alias on the having clause it will not work because at the time of evaluating the having clause this column does not exist. See [SQL order of execution](#)

13.5 Having Vs Where

HAVING works on the RESULT GROUP whereas WHERE works on the SELECT columns and not the result group.

```
SELECT
movie_lang,
SUM(movie_length)
FROM movies
GROUP BY movie_lang
HAVING SUM(movie_length) > 200
ORDER BY 2 DESC;
```

	movie_lang character varying (20)	sum bigint
1	English	4824
2	Chinese	609
3	Japanese	446
4	Portuguese	285

Get the movie languages where their sum total movie length is greater than 200.

Note that first SQL is summing the movie length then performing the HAVING selection on the SUM results. **HAVING is performed on the AGGREGATE RESULT GROUP.**

```
SELECT
movie_lang,
SUM(movie_length)
FROM movies
GROUP BY movie_lang
WHERE SUM(movie_length) > 200
ORDER BY 2 DESC;
ERROR: syntax error at or near "WHERE"
LINE 6: WHERE SUM(movie_length) > 200
      ^
SQL state: 42601 Character: 71
```

Can I use WHERE clause on an Aggregate? - **NO**

```
SELECT
movie_lang,
SUM(movie_length)
FROM movies
WHERE movie_length > 100
GROUP BY movie_lang
ORDER BY 2 DESC;
```

WHERE clause is used with the SELECTed columns.

	movie_lang character varying (20)	sum bigint
1	English	4551
2	Chinese	510
3	Japanese	446
4	Portuguese	285
5	German	165
6	Korean	130
7	Swedish	128

13.5 Handling Null Values with Group By using Coalesce

```
CREATE TABLE employee_test (  
  employee_id SERIAL PRIMARY KEY,  
  employee_name VARCHAR(100),  
  department VARCHAR(100),  
  salary INT  
);
```

```
INSERT INTO employee_test  
(employee_name, department,  
salary)  
VALUES  
( 'John', 'Finance', 2500),  
( 'Mary', NULL, 3000),  
( 'Adam', NULL, 4000),  
( 'Bruce', 'Finance', 4000),  
( 'Linda', 'IT', 5000),  
( 'Megan', 'IT', 4000);
```

```
SELECT  
department,  
Count(salary) AS "total employees"  
FROM employee_test  
GROUP BY department  
ORDER BY department
```

	department character varying (100) 🔒	total employees bigint 🔒
1	Finance	2
2	IT	2
3	[null]	2

```
SELECT * FROM employee_test ORDER BY department;
```

	employee_id [PK] integer ✎	employee_name character varying (100) ✎	department character varying (100) ✎	salary integer ✎
1	1	John	Finance	2500
2	4	Bruce	Finance	4000
3	5	Linda	IT	5000
4	6	Megan	IT	4000
5	2	Mary	[null]	3000
6	3	Adam	[null]	4000

```
SELECT  
COALESCE(department, 'NO DEPARTMENT') AS department,  
Count(salary) AS "total employees"  
FROM employee_test  
GROUP BY department  
ORDER BY department
```

	department character varying 🔒	total employees bigint 🔒
1	Finance	2
2	IT	2
3	NO DEPARTMENT	2

How many employee's
in each department?

If there are null values in a column I can use coalesce to either display the column source value or a chosen value **COALESCE(source, 'chosen value')**

14. Joining Multiple Tables

[14.1 Inner Joins](#)

[14.2 Inner Joins with Using](#)

[14.3 Inner Joins with Filter Data](#)

[14.4 Inner Joins with Different Data Type Columns](#)

[14.5 Left Joins](#)

[14.6 Right Joins](#)

[14.7 Full Joins](#)

[14.8 Joining Multiple Tables](#)

[14.9 Self Joins](#)

[14.10 Cross Joins](#)

[14.11 Natural Joins](#)

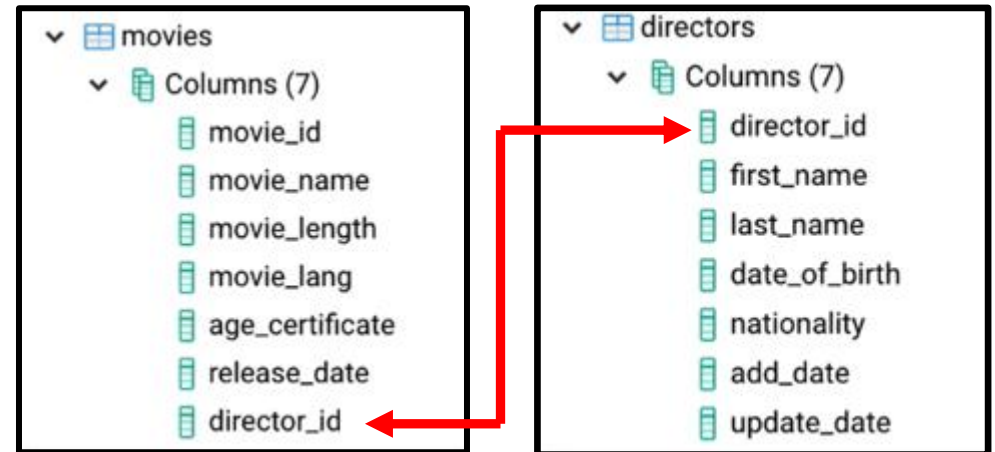
[14.13 Append Tables with different Columns](#)

[14.14 On Vs Where](#)

14.1 Inner Joins

PostgreSQL join is used to combine/attach columns from one or more tables based on the values of the COMMON COLUMNS between tables. These COMMON COLUMNS are generally PRIMARY KEY columns of the first table and FOREIGN KEY columns of the second table.

```
SELECT *
FROM table_a
JOIN table_a
ON table_a.key_column =
table_b.foreign_key_column;
```



```
SELECT
movie_id,
movie_name,
director_id
FROM movies
INNER JOIN directors
ON movies.director_id = directors.director_id;
ERROR: column reference "director_id" is ambiguous
LINE 4: director_id
          ^
SQL state: 42702 Character: 30
```

The column is ambiguous because 'director_id' could refer to the column from movies table or the column from directors table.

Specify the table.column_name in the select clause to identify which table the column comes from

```
SELECT
movies.movie_id,
movies.movie_name,
movies.director_id
FROM movies
INNER JOIN directors
ON movies.director_id =
directors.director_id;
```

The motive for using joins is to combine column data from one or more tables in the result table.

```
SELECT
movies.movie_id,
movies.movie_name,
movies.director_id,
directors.first_name
FROM movies
INNER JOIN directors
ON movies.director_id = directors.director_id;
```

Long way to write
column names

```
SELECT
mv.movie_id,
mv.movie_name,
mv.director_id,
d.first_name
FROM movies mv
INNER JOIN directors d
ON mv.director_id = d.director_id;
```

Alias table names

Writing tablename.column_name can get really tedious so it is better to alias the columns in the select clause. Note how I alias the table movies to 'mv' in the from clause and I alias directors table as 'd' in the inner clause Join. I can now use the aliases when referencing the columns in the select clause and the on clause.

The ON clause is to link the common column

	movie_id integer	movie_name character varying (100)	director_id integer	first_name character varying (150)
1	1	A Clockwork Orange	13	Stanley
2	2	Apocalypse Now	9	Francis
3	3	Battle Royale	10	Kinji
4	4	Blade Runner	27	Ridley
5	5	Chunking Express	35	Kar Wai

```
SELECT
mv.movie_id,
mv.movie_name,
mv.movie_lang,
mv.director_id,
d.first_name
FROM movies mv
INNER JOIN directors d
ON mv.director_id = d.director_id
WHERE mv.movie_lang = 'English';
```

	movie_id integer	movie_name character varying (100)	movie_lang character varying (20)	director_id integer	first_name character varying (150)
1	46	There Will Be Blood	English	2	Paul
2	40	The Darjeeling Limited	English	3	Wes
3	30	Rushmore	English	3	Wes

I can use a where clause to filter results by a column

```
SELECT
mv.movie_id,
mv.movie_name,
mv.movie_lang,
mv.director_id,
d.first_name
FROM movies mv
INNER JOIN directors d
ON mv.director_id = d.director_id
WHERE mv.movie_lang = 'English'
AND d.director_id = 3;
```

	movie_id integer	movie_name character varying (100)	movie_lang character varying (20)	director_id integer	first_name character varying (150)
1	15	Grand Budapest Hotel	English	3	Wes
2	30	Rushmore	English	3	Wes
3	40	The Darjeeling Limited	English	3	Wes

I can use a where clause to filter results by a column and use AND to filter by two columns.

```
SELECT
mv.*,
d.*
FROM movies mv
INNER JOIN directors d
ON mv.director_id = d.director_id
WHERE mv.movie_lang = 'English'
AND d.director_id = 3;
```

Alias.* will get all columns from a table. We would not use this in a production environment as a query to present to to end user or in a model but for database admin it is sometimes useful to check the data.

	movie_id integer	movie_name character varying (100)	movie_length integer	movie_lang character varying (20)	age_certificate character varying (10)	release_date date	director_id integer	director_id integer	first_name character varying (150)	last_name character varying (150)	date_of_birth date	nationality character varying (20)	add_date date	update_date date
1	15	Grand Budapest Hotel	117	English	PG	2014-07-03	3	3	Wes	Anderson	1969-05-01	American	[null]	[null]
2	30	Rushmore	104	English	12	1998-11-12	3	3	Wes	Anderson	1969-05-01	American	[null]	[null]
3	40	The Darjeeling Limited	119	English	PG	2007-09-29	3	3	Wes	Anderson	1969-05-01	American	[null]	[null]

14.2 Inner Joins with Using

We use USING only when joining tables that have the SAME column names rather than ON!!

```
SELECT
movies.movie_id,
movies.movie_name,
movies.movie_lang,
directors.director_id,
directors.first_name
FROM movies
INNER JOIN directors USING (director_id);
```

Syntax:

```
SELECT
Table1.column1,
Table2.column1
FROM table_a
INNER JOIN table2 using (column1)
```

	movie_id integer	movie_name character varying (100)	movie_lang character varying (20)	director_id integer	first_name character varying (150)
1	1	A Clockwork Orange	English	13	Stanley
2	2	Apocalypse Now	English	9	Francis
3	3	Battle Royale	Japanese	10	Kinii

```
SELECT
movies.movie_name,
movies_revenues.revenues_domestic,
movies_revenues.revenues_international
FROM movies
INNER JOIN movies_revenues USING (movie_id);
```

Note that when I utilise USING for Joins I am not using table aliasing so the column identities are written in full as table_name.column_name

	movie_name character varying (100)	revenues_domestic numeric (10,2)	revenues_international numeric (10,2)
1	The Wizard of Oz	22.20	1.30
2	Gone with the Wind	199.40	201.20
3	Mary Poppins	102.10	[null]



```
SELECT
movies.movie_name,
movies_revenues.revenues_domestic,
movies_revenues.revenues_international,
directors.first_name || ' ' || directors.last_name AS "director"
FROM movies
INNER JOIN movies_revenues USING (movie_id)
INNER JOIN directors USING (director_id);
```

I can combine data from three tables with the USING keyword

	movie_name character varying (100)	revenues_domestic numeric (10,2)	revenues_international numeric (10,2)	director text
1	The Wizard of Oz	22.20	1.30	Victor Fleming
2	Gone with the Wind	199.40	201.20	Victor Fleming
3	Mary Poppins	102.10	[null]	Robert Stevenson
4	The Sound of Music	150.70	[null]	Robert Wise

14.3 Inner Joins with Filter Data

```
SELECT
movies.movie_name,
movies_revenues.revenues_domestic,
directors.first_name || ' ' || directors.last_name AS "director"
FROM movies
INNER JOIN movies_revenues USING (movie_id)
INNER JOIN directors USING (director_id)
WHERE movies.movie_lang = 'Japanese';
```

USING

```
SELECT
mv.movie_name,
r.revenues_domestic,
d.first_name || ' ' || d.last_name AS "director"
FROM movies mv
INNER JOIN directors d ON mv.director_id = d.director_id
INNER JOIN movies_revenues r ON mv.movie_id = r.movie_id
WHERE mv.movie_lang = 'Japanese';
```

ON

Get movie name, director name & domestic revenues for all Japanese movies

	movie_name character varying (100) 🔒	revenues_domestic numeric (10,2) 🔒	director text 🔒
1	Battle Royale	[null]	Kinji Fukasaku
2	Ponyo	15.10	Hayao Miyaz...
3	Spirited Away	11.10	Hayao Miyaz...
4	Cold Fish	[null]	Sion Sono

```
SELECT
mv.movie_name,
d.first_name || ' ' || d.last_name AS "director",
r.revenues_domestic,
r.revenues_international
FROM movies mv
INNER JOIN directors d ON mv.director_id = d.director_id
INNER JOIN movies_revenues r ON mv.movie_id = r.movie_id
WHERE mv.movie_lang IN('English','Japanese','Chinese')
AND r.revenues_domestic > 100
ORDER BY mv.movie_lang;
```

Get movie name, director name, domestic and international revenues for all English, Japanese and Chinese movies.

	movie_name character varying (100)	director text	revenues_domestic numeric (10,2)	revenues_international numeric (10,2)
1	Crouching Tiger Hidden Drag...	Ang Lee	128.10	85.10
2	Gone with the Wind	Victor Fleming	199.40	201.20
3	Ten Cents a Reel	John S. Robertson	100.10	100.10

Get movie name, director name and movie language for all top five total revenues movies.

```
SELECT
mv.movie_name,
d.first_name || ' ' || d.last_name AS "director",
mv.movie_lang,
(r.revenues_domestic + r.revenues_international) AS "total_revenues"
FROM movies mv
INNER JOIN directors d ON mv.director_id = d.director_id
INNER JOIN movies_revenues r ON mv.movie_id = r.movie_id
ORDER BY total_revenues DESC NULLS LAST
LIMIT 5;
```

	movie_name character varying (100)	director text	movie_lang character varying (20)	total_revenues numeric
1	Titanic	James Cameron	English	2187.30
2	Spider-Man 3	Sam Raimi	English	891.00
3	Spider-Man	Sam Raimi	English	822.20
4	Spider-Man 2	Sam Raimi	English	784.50
5	Star Wars: A New Hope	George Lucas	English	775.40


```

SELECT
mv.movie_name,
d.first_name || ' ' || d.last_name AS "director",
mv.movie_lang,
(r.revenues_domestic + r.revenues_international) AS "total_revenues"
FROM movies mv
INNER JOIN directors d ON mv.director_id = d.director_id
INNER JOIN movies_revenues r ON mv.movie_id = r.movie_id
WHERE mv.release_date BETWEEN '2005-01-01' AND '2008-12-31'
ORDER BY total_revenues DESC NULLS LAST
LIMIT 10;

```

	movie_name character varying (100) 🔒	director text 🔒	movie_lang character varying (20) 🔒	total_revenues numeric 🔒
1	Spider-Man 3	Sam Raimi	English	891.00
2	V for Vendetta	James McTeigue	English	133.70
3	Pans Labyrinth	Guillermo del Toro	Spanish	84.20
4	The Lives of Others	Florian Henckel von Donnersmar...	German	77.40
5	There Will Be Blood	Paul Anderson	English	75.70
6	The Darjeeling Limited	Wes Anderson	English	35.10
7	Oldboy	Chan-wook Park	Korean	14.90
8	Let the Right One In	Tomas Alfredson	Swedish	11.20
9	City of Men	Paulo Morelli	Portuguese	2.50

Get the movie name, director name and total revenues for the 10 most profitable movies between 2005 and 2008.

14.4 Inner Joins with Different Data Type Columns

```
CREATE TABLE t1 (test INT);  
CREATE TABLE t2 (test VARCHAR(10));
```

```
SELECT  
* FROM t1  
INNER JOIN t2 ON t1.test::VARCHAR = t2.test;  
ERROR: operator does not exist: integer = character varying  
LINE 6: INNER JOIN t2 ON t1.test = t2.test;  
                                ^
```

HINT: No operator matches the given name and argument types. You might need to add explicit type casts.
SQL state: 42883 Character: 110

Can I join two tables on columns with different data types? **NO & YES!!!**

```
SELECT  
* FROM t1  
INNER JOIN t2 ON t1.test::VARCHAR = t2.test;
```

	test integer	test character varying (10)

Here I am casting the left hand table as VARCHAR to make the join work

```
SELECT  
* FROM t1  
INNER JOIN t2 ON t1.test = CAST(t2.test AS INT);
```

	test integer	test character varying (10)

Here I am casting the right hand table as INT to make the join work

```
INSERT INTO t1 (test) VALUES (1),(2);  
INSERT INTO t2 (test) VALUES (1),(2);  
INSERT INTO t2 (test) VALUES ('aa'),('bb');
```

```
SELECT  
* FROM t1  
INNER JOIN t2 ON t1.test = CAST(t2.test AS INT);
```

	test integer	test character varying (10)
1	1	1
2	2	2

**USING CAST I
CAN JOIN ON
DIFFERENT
DATA TYPES**

The join
returns data

14.5 Left Joins

```
SELECT
Table1.column1,
Table2.column1
FROM table1
LEFT JOIN table2
ON table1.column1 = Table2.column1;
```

Returns every row from the left table PLUS rows that match the values in the joined column from the right table. When a left table row does not have a match in the right table the results show no values from the right table.

```
CREATE TABLE left_products (
    product_id SERIAL PRIMARY KEY,
    product_name VARCHAR(100)
);

CREATE TABLE right_products (
    product_id SERIAL PRIMARY KEY,
    product_name VARCHAR(100)
);

INSERT INTO left_products (product_id, product_name) VALUES
(1, 'Computers'), (2, 'Laptops'), (3, 'Monitors'), (5, 'Mics');

INSERT INTO Right_products (product_id, product_name) VALUES
(1, 'Computers'), (2, 'Laptops'), (3, 'Monitors'), (4, 'Pen'), (7, 'Papers');
```

Left join selects all the rows from the left table AND matching rows from the right table on the joining column.

```
SELECT *
FROM left_products
LEFT JOIN right_products
ON left_products.product_id =
right_products.product_id;
```

	product_id integer	product_name character varying (100)	product_id integer	product_name character varying (100)
1	1	Computers	1	Computers
2	2	Laptops	2	Laptops
3	3	Monitors	3	Monitors
4	5	Mics	[null]	[null]

```
SELECT
d.first_name || ' ' || d.last_name AS "director",
mv.movie_name
FROM directors d
LEFT JOIN movies mv ON mv.director_id = d.director_id
ORDER BY 1;
```

Joining 'directors' to 'movies'
with a left join produces 53 rows.

	director text	movie_name character varying (100)
1	Ang Lee	Crouching Tiger Hidden Dragon
2	Ang Lee	Life of Pi
3	Bruce Lee	Way of the Dragon
4	Chan-wook Park	Oldboy

```
SELECT
d.first_name || ' ' || d.last_name AS "director",
mv.movie_name
FROM movies mv
LEFT JOIN directors d ON d.director_id = mv.director_id
ORDER BY 1;
```

Joining 'movies' to 'directors'
with a left join also produces 53
rows.

```
INSERT INTO directors (first_name, last_name, date_of_birth, nationality) VALUES
('james', 'David', '2010-01-01', 'American');
```

```
SELECT
d.first_name || ' ' || d.last_name AS "director",
mv.movie_name
FROM directors d
LEFT JOIN movies mv ON mv.director_id = d.director_id
ORDER BY d.date_of_birth DESC;
```

Joining 'directors' to 'movies'
with a left join now produces 54
rows because I have added a
director.

	director text	movie_name character varying (100)
1	james David	[null]
2	Richard Ayoade	Submarine
3	Florian Henckel von Donnersmarck	The Lives of Others

```
SELECT
d.first_name || ' ' || d.last_name AS "director",
mv.movie_name
FROM movies mv
LEFT JOIN directors d ON d.director_id = mv.director_id
ORDER BY d.date_of_birth DESC;
```

Joining 'movies' to 'directors' with a left join only produces 53 rows because the recently added director does not have any movies.

```
SELECT
d.first_name || ' ' || d.last_name AS "director",
mv.movie_name
FROM directors d
LEFT JOIN movies mv ON mv.director_id = d.director_id
WHERE mv.movie_lang IN('English','Chinese');
```

Get all movie names and directors for English and Chinese movies.

	director text	movie_name character varying (100)
1	Stanley Kubrick	A Clockwork Orange
2	Francis Ford Coppola	Apocalypse Now
3	Ridley Scott	Blade Runner
4	Kar Wai Wong	Chungking Express

```
SELECT
d.first_name || ' ' || d.last_name AS "director",
COUNT(mv.movie_name) AS "total movies"
FROM directors d
LEFT JOIN movies mv ON mv.director_id = d.director_id
GROUP BY director
ORDER BY COUNT(mv.movie_name) DESC;
```

Get a list of directors with a movie count per director.

	director text	total movies bigint
1	Stanley Kubrick	3
2	Martin Scorsese	3
3	George Lucas	3
4	Sam Raimi	3
5	Wes Anderson	3
6	Kar Wai Wong	2
7	Apocalypse Now	1

```
SELECT
mv.movie_name,
mv.age_certificate,
d.first_name || ' ' || d.last_name AS "director"
FROM directors d
LEFT JOIN movies mv ON mv.director_id = d.director_id
WHERE d.nationality
IN('American','Japanese','Chinese')
ORDER BY director;
```

Get a list of movies with age certification for all directors that are American, Chinese or Japanese

	movie_name character varying (100)	age_certificate character varying (10)	director text
1	Crouching Tiger Hidden Drag...	12	Ang Lee
2	Life of Pi	PG	Ang Lee
3	Way of the Dragon	12	Bruce Lee
4	Apocalypse Now	15	Francis Ford Coppola

```
SELECT
d.first_name || ' ' || d.last_name AS "director",
SUM(r.revenues_domestic + r.revenues_international) AS "total_revenues"
FROM directors d
LEFT JOIN movies mv ON mv.director_id = d.director_id
LEFT JOIN movies_revenues r ON r.movie_id = mv.movie_id
GROUP BY director
HAVING SUM(r.revenues_domestic + r.revenues_international) > 0
ORDER BY 2 DESC NULLS LAST;
```

Get all the total revenues for each film for each director.

	director text	total_revenues numeric
1	Sam Raimi	2497.70
2	James Cameron	2187.30
3	George Lucas	1788.80
4	Ang Lee	822.20

14.6 Right Joins

```
SELECT  
Table1.column1,  
Table2.column1  
FROM table1  
LEFT JOIN table2  
ON table1.column1 = Table2.column1;
```

Returns every row from the right table PLUS rows that match the values in the joined column from the left table. When a right table row does not have a match in the left table the results show no values from the left table.

```
SELECT *  
FROM left_products  
RIGHT JOIN right_products  
ON left_products.product_id =  
right_products.product_id;
```

	product_id integer	product_name character varying (100)	product_id integer	product_name character varying (100)
1	1	Computers	1	Computers
2	2	Laptops	2	Laptops
3	3	Monitors	3	Monitors
4	[null]	[null]	4	Pen
5	[null]	[null]	7	Papers

```
SELECT  
mv.movie_name,  
d.first_name || ' ' || d.last_name AS "director"  
FROM movies mv  
LEFT JOIN directors d ON d.director_id = mv.director_id;
```

	movie_name character varying (100)	director text
1	A Clockwork Orange	Stanley Kubrick
2	Apocalypse Now	Francis Ford Coppola
3	Battle Royale	Kinji Fukasaku
4	Blade Runner	Ridley Scott

List all the movies with directors names and movie names

The movies table needs to be the right table because we want all movies then find matching directors.

```
SELECT
mv.movie_name,
d.first_name || ' ' || d.last_name AS "director"
FROM movies mv
RIGHT JOIN directors d ON d.director_id = mv.director_id;
```

List all English and Chinese movies with directors names and movie names

	movie_name character varying (100)	movie_lang character varying (20)	director text
1	A Clockwork Orange	English	Stanley Kubrick
2	Apocalypse Now	English	Francis Ford Coppola
3	Blade Runner	English	Ridley Scott
4	Chungking Express	Chinese	Kar Wai Wong
5	Crouching Tiger Hidden Dragon	Chinese	Ang Lee

```
SELECT
d.first_name || ' ' || d.last_name AS "director",
COUNT(*) AS "movies_count"
FROM movies mv
RIGHT JOIN directors d ON d.director_id = mv.director_id
GROUP BY director
ORDER BY 2 DESC;
```

Count all movies for each director.

	director text	movies_count bigint
1	Sam Raimi	3
2	Martin Scorsese	3
3	George Lucas	3
4	Wes Anderson	3


```
SELECT
mv.movie_name,
d.first_name || ' ' || d.last_name AS "director"
FROM movies mv
RIGHT JOIN directors d ON d.director_id = mv.director_id;
```

List all English and Chinese movies with directors names and movie names

	movie_name character varying (100)	director text
1	A Clockwork Orange	Stanley Kubrick
2	Apocalypse Now	Francis Ford Coppola
3	Battle Royale	Kinji Fukasaku
4	Blade Runner	Ridley Scott
5	Chungking Express	Kar Wai Wong

```
SELECT
d.first_name || ' ' || d.last_name AS "director",
SUM(r.revenues_domestic + r.revenues_international) AS "total_revenues"
FROM directors d
RIGHT JOIN movies mv ON d.director_id = mv.director_id
LEFT JOIN movies_revenues r ON r.movie_id = mv.movie_id
GROUP BY director
HAVING SUM(r.revenues_domestic + r.revenues_international) > 100
ORDER BY 2 DESC;
```

List all total revenues done by each film for each director for total revenues greater than 100.

	director text	total_revenues numeric
1	Sam Raimi	2497.70
2	James Cameron	2187.30
3	George Lucas	1788.80
4	Ang Lee	822.20
5	Robert Zemeckis	678.40

14.7 Full Joins

```
SELECT  
Table1.column1,  
Table2 column1  
FROM table1  
FULL JOIN table2  
ON table1.column1 = Table2.column1;
```

Full Join returns every row from all the row tables

```
SELECT *  
FROM left_products  
FULL JOIN right_products ON left_products.product_id = right_products.product_id;
```

	product_id integer	product_name character varying (100)	product_id integer	product_name character varying (100)
1	1	Computers	1	Computers
2	2	Laptops	2	Laptops
3	3	Monitors	3	Monitors
4	5	Mics	[null]	[null]
5	[null]	[null]	4	Pen
6	[null]	[null]	7	Papers

	product_id integer	product_name character varying (100)	product_id integer	product_name character varying (100)
1	1	Computers	1	Computers
2	2	Laptops	2	Laptops
3	3	Monitors	3	Monitors
4	4	Pen	[null]	[null]
5	7	Papers	[null]	[null]
6	[null]	[null]	5	Mics

```
SELECT *  
FROM right_products  
FULL JOIN left_products ON left_products.product_id = right_products.product_id;
```

Note that full join includes all rows from both the left and right tables even if there is no match on the joining column.

```
SELECT
d.first_name || ' ' || d.last_name AS "director",
mv.movie_name
FROM directors d
FULL JOIN movies mv ON d.director_id = mv.director_id
ORDER BY director;
```

	director text	movie_name character varying (100)
45	Terry Jones	Life of Brian
46	Tomas Alfredson	Let the Right One In
47	Tony Scott	Top Gun
48	Victor Fleming	Gone with the Wind
49	Victor Fleming	The Wizard of Oz
50	Wes Anderson	The Darjeeling Limited

This join query produces 54 rows. An inner join would produce 53 rows because there is one director that does not have a movie.

```
SELECT
d.first_name || ' ' || d.last_name AS "director",
mv.movie_name
FROM movies mv
FULL JOIN directors d ON d.director_id = mv.director_id
ORDER BY director;
```

Swapping the table around still produces 54 rows

```

SELECT
d.first_name || ' ' || d.last_name AS "director",
mv.movie_name
FROM movies mv
FULL JOIN directors d ON d.director_id =
mv.director_id
WHERE mv.movie_lang IN('English','Japanese','Chinese')
ORDER BY director;

```

I can add a where clause to filter the results down to 47 rows.

	director text	movie_name character varying (100)
1	Ang Lee	Life of Pi
2	Ang Lee	Crouching Tiger Hidden Dragon
3	Bruce Lee	Way of the Dragon
4	Francis Ford Coppola	Apocalypse Now
5	George Lucas	Star Wars: Empire Strikes Back
6	George Lucas	Star Wars: Return of the Jedi
7	George Lucas	Star Wars: A New Hope

```

SELECT
Table1.column1,
Table2 column1,
Table3 column1
FROM table1
JOIN table2 ON table1.column1 = Table2.column1
JOIN table3 ON table1.column1 = Table3.column1;

```

14.8 Joining Multiple Tables

It is possible to join multiple tables together via JOIN statement.

Does the order of the joining tables matter?

Not really. Only the column order of the result table changes. The 'from' table will be the first columns followed by the columns from the joined tables in the order in which they were joined.

```

SELECT *
FROM movies mv
JOIN directors d ON mv.director_id = d.director_id
JOIN movies_revenues r ON mv.movie_id = r.movie_id

SELECT *
FROM directors d
JOIN movies mv ON d.director_id = mv.director_id
JOIN movies_revenues r ON mv.movie_id = r.movie_id;

```

```

SELECT *
FROM actors ac
JOIN movies_actors ma ON ma.actor_id = ac.actor_id
JOIN movies mv ON mv.movie_id = ma.movie_id
JOIN directors d ON d.director_id = mv.director_id
JOIN movies_revenues r ON mv.movie_id = r.movie_id

```

These multi table join query produces 70 rows.

```

SELECT *
FROM actors ac
INNER JOIN movies_actors ma ON ma.actor_id = ac.actor_id
INNER JOIN movies mv ON mv.movie_id = ma.movie_id
INNER JOIN directors d ON d.director_id = mv.director_id
INNER JOIN movies_revenues r ON mv.movie_id = r.movie_id

```

INNER JOIN = JOIN. Inner join is the same as join.

14.9 Self Joins

```
SELECT  
Column list  
FROM table_name t1  
INNER JOIN table_name t1 ON t1.column = t2.column
```

Self Joins allow the comparing of rows within the SAME table. Normal use of a self join is to query hierarchical data or to compare rows within the same table.

```
SELECT *  
FROM left_products t1  
INNER JOIN right_products t2 ON t1.product_id = t2.product_id
```

	product_id integer	product_name character varying (100)	product_id integer	product_name character varying (100)
1	1	Computers	1	Computers
2	2	Laptops	2	Laptops
3	3	Monitors	3	Monitors

```
SELECT *  
FROM left_products t1  
INNER JOIN right_products t2 ON CAST(t1.product_id AS VARCHAR) = t2.product_name
```

	product_id integer	product_name character varying (100)	product_id integer	product_name character varying (100)

I can cast a column to compare it with a different data type but the query will return empty because there is no comparison.

```
SELECT *  
FROM directors t1  
INNER JOIN directors t2 ON t1.director_id = t2.director_id;
```

This will compare director ID with director ID within the same table.

```
SELECT
t1.movie_name,
t2.movie_name,
t1.movie_length,
t2.movie_length
FROM movies t1
INNER JOIN movies t2 ON t1.movie_length = t2.movie_length;
```

Self Join find all pair if movies that have the same movie length.

	movie_name character varying (100)	movie_name character varying (100)	movie_length integer	movie_length integer
1	A Clockwork Orange	Spider-Man 3	112	112
2	A Clockwork Orange	A Clockwork Orange	112	112
3	Apocalypse Now	There Will Be Blood	168	168
4	Apocalypse Now	Apocalypse Now	168	168
5	Battle Royale	Battle Royale	111	111
6	Blade Runner	Top Gun	121	121

But we have repeats of the same movie title.

```
SELECT
t1.movie_name,
t2.movie_name,
t1.movie_length,
t2.movie_length
FROM movies t1
INNER JOIN movies t2 ON t1.movie_length = t2.movie_length
AND t1.movie_name <> t2.movie_name
ORDER BY t1.movie_length DESC;
```

I can refine the query to say that movie_name from t1 is not equal to the movie_name from t2

	movie_name character varying (100)	movie_name character varying (100)	movie_length integer	movie_length integer
1	Star Wars: A New Hope	Leon	123	123
2	Never Let Me Go	Taxi Driver	117	117
3	A Clockwork Orange	Spider-Man 3	112	112
4	V for Vendetta	City of Men	140	140
5	Eyes Wide Shut	Oldboy	130	130
6	Top Gun	Blade Runner	121	121

I am not using a where clause but pushing the not equal onto the on clause

```
SELECT
t1.movie_name,
t2.director_id
FROM movies t1
INNER JOIN movies t2 ON t1.director_id = t2.movie_id
ORDER BY t2.director_id, t1.movie_name;
```

Each film has a director with the 'director_id' being of a hierarchical nature. I can use a self join to compare movies rows with movies rows on movie name and include the director.

	movie_name character varying (100)	director_id integer
1	City of God	1
2	Cold Fish	3
3	Crouching Tiger Hidden Dragon	3
4	Life of Pi	3
5	V for Vendetta	5
6	Goodfellas	7
7	Raging Bull	7

14.10 Cross Joins

In a CROSS JOIN query, the result (also known as Cartesian product) lines up each row in the left table with each row in the right table to present all possible combinations of rows.

Suppose you need to perform a CROSS JOIN of two tables, table1 and table2. if table1 has n rows and table2 has m rows, the result set will have $n * m$ rows.

For example, the table1 has 1000 rows and table2 has 1000 rows then the result set will have 1000 x 1000 rows (1 million rows)

Unlike INNER JOIN, LEFT JOIN and RIGHT JOIN, CROSS JOIN clause does not have a join predicate.

```
SELECT
Column list
FROM table1
CROSS JOIN table2
```



```
SELECT *
FROM left_products
CROSS JOIN right_products;
```

The cross join produces 20 rows of data because the left_products table has 4 rows and the right_products table has 5 rows. (4 x 5 = 20)

	product_id integer	product_name character varying (100)	product_id integer	product_name character varying (100)
1	1	Computers	1	Computers
2	1	Computers	2	Laptops
3	1	Computers	3	Monitors
4	1	Computers	4	Pen
5	1	Computers	7	Papers
6	2	Laptops	1	Computers

If I swap the tables around then the results will be different because the from table is being compared with the cross join table.

```
SELECT *
FROM left_products, right_products;
```

This method of selecting from two tables will produce the same results as the above cross join.

```
SELECT *
FROM table1
INNER JOIN table2 ON true;
```

```
SELECT *
FROM left_products
INNER JOIN right_products ON true;
```

This syntax will also produce a cross join.

```
SELECT * FROM actors; --> 147 rows
SELECT * FROM directors; --> 38 rows
-- CROSS JOIN = 147 * 38 = 5586 rows
```

```
SELECT *
FROM actors
CROSS JOIN directors;
```

Showing rows: 5001 to 5586  Page No: 6 of 6					
g (150)	gender character (1)	date_of_birth date	update_date date	director_id integer	first cha
	M	1960-04-04	[null]	23	Ch
	M	1960-04-04	[null]	24	Sa
	M	1960-04-04	[null]	25	Ma
	M	1960-04-04	[null]	26	Ma

Cross Joins should be avoided on large tables of 1000 or more rows because it will chew up server memory returning the result.

14.11 Natural Joins

A natural join is a join that creates an **implicit join based on the same column names** in the joined tables. A natural join can be an inner join, left join or right join. If you do not specify a join explicitly then PostGreSQL will use an INNER JOIN by default.

```
SELECT  
Column list  
FROM table1  
NATURAL [INNER, LEFT, RIGHT] JOIN table2;
```

```
SELECT *  
FROM left_products  
NATURAL JOIN right_products;
```

	product_id [PK] integer	product_name character varying (100)
1	1	Computers
2	2	Laptops
3	3	Monitors

```
SELECT *  
FROM left_products  
NATURAL INNER JOIN right_products;
```

	product_id [PK] integer	product_name character varying (100)
1	1	Computers
2	2	Laptops
3	3	Monitors

A natural join will join ALL the common columns

```
SELECT *  
FROM left_products  
NATURAL LEFT JOIN right_products;
```

	product_id [PK] integer	product_name character varying (100)
1	1	Computers
2	2	Laptops
3	3	Monitors
4	5	Mics

```
SELECT *  
FROM left_products  
NATURAL RIGHT JOIN right_products;
```

	product_id [PK] integer	product_name character varying (100)
1	1	Computers
2	2	Laptops
3	3	Monitors
4	4	Pen
5	7	Papers

```
SELECT *  
FROM movies  
NATURAL JOIN directors;
```

This join will produce a table of 53 rows. It will join on 'director_id' which is the common column from both tables. Because I have implicitly set the join type it is performing an inner join by default.

14.13 Append Tables with different Columns

We have two tables:

table1 with columns (add_date, col1, col2, col3)

table2 with columns (add_date, col1,col2,col3,col4,col5)

We want to combine the two tables, but making sure to take the data from FIRST TABLE, table 1.

```
CREATE TABLE table1 (  
    add_date DATE,  
    col1 INT,  
    col2 INT,  
    col3 INT  
);
```

```
INSERT INTO table1 (add_date, col1, col2, col3) VALUES  
( '2020-01-01', 1, 2, 3),  
( '2020-01-02', 4, 5, 6);
```

```
CREATE TABLE table2 (  
    add_date DATE,  
    col1 INT,  
    col2 INT,  
    col3 INT,  
    col4 INT,  
    col5 INT  
);
```

```
INSERT INTO table2 (add_date, col1, col2, col3, col4, col5) VALUES  
( '2020-01-01', NULL, 7, 8, 9, 10),  
( '2020-01-02', 11, 12,13,14,15),  
( '2020-01-05', 16, 17,18,19,20);
```

```
SELECT COALESCE(100,2);
```

	coalesce integer
1	100

The coalesce function takes the first value if it is not null or it will use the second value.

```
SELECT  
COALESCE(t1.add_date, t2.add_date) AS "add_date",  
COALESCE(t1.col1, t2.col1) AS "col1",  
COALESCE(t1.col2, t2.col2) AS "col2",  
COALESCE(t1.col3, t2.col3) AS "col3",  
t2.col4,  
t2.col5  
FROM table1 t1  
FULL OUTER JOIN table2 t2 ON (t1.add_date = t2.add_date);
```

	add_date date	col1 integer	col2 integer	col3 integer	col4 integer	col5 integer
1	2020-01-01	1	2	3	9	10
2	2020-01-02	4	5	6	14	15
3	2020-01-05	16	17	18	19	20

14.14 On Vs Where

The function of ON and WHERE clauses in the various types of the joins like INNER, OUTER, LEFT, RIGHT and more might be a bit confusing so bear in mind the following facts:

1. The ON clause is part of the INNER, LEFT, RIGHT and FULL JOINS. THE CROSS JOIN and UNION JOIN do not have an ON clause because neither of them does any filtering of the data.
2. The ON clause in an INNER JOIN is logically equivalent to a WHERE clause. I.e. the same condition could be specified either in an ON clause or a WHERE clause.
3. The ON clause is outer join i.e. LEFT, RIGHT and FULL joins are different from WHERE clauses.
 - a. The WHERE clause simply filters the rows returned by the FROM clause. Rows rejected by the filter are not included in the result.
 - b. The ON clause in an OUTER join first filters the rows of a cross join product and then includes the rejected rows, extended with null values.

15. Combining Queries Together

[15.1 Combine Results Sets with Union](#)

[15.2 Union with Filters and Conditions](#)

[15.3 Union Tables with Different Number of Columns](#)

[15.4 Intersect with Tables](#)

[15.5 Except with Tables](#)

15.1 Combine Results Sets with Union

The UNION operator combines results sets from two or more SELECT statements into a single result set.

To be able to use Union:

The order and number of the columns in the select list of all queries must be the same.

The data types must be compatible also

```
SELECT column1, column2
FROM table1
UNION
SELECT column1, column2
FROM table2
```

Do we get duplicate records when we use UNION?

```
INSERT INTO right_products (product_id, product_name) VALUES
(10, 'Pen');
```

```
SELECT product_id, product_name
FROM left_products
UNION
SELECT product_id, product_name
FROM right_products;
```

```
SELECT product_id, product_name
FROM left_products
UNION
SELECT product_id, product_name
FROM right_products;
```

The union does not return duplicates. Note how the Pen has ID 4 in one table and ID 10 in another table so is not the same record

	product_id integer	product_name character varying (100)
1	1	Computers
2	5	Mics
3	7	Papers
4	2	Laptops
5	3	Monitors
6	4	Pen

	product_id integer	product_name character varying (100)
1	10	Pen
2	1	Computers
3	5	Mics
4	7	Papers
5	2	Laptops
6	3	Monitors
7	4	Pen

	product_id integer	product_name character varying (100)
1	1	Computers
2	2	Laptops
3	3	Monitors
4	5	Mics
5	1	Computers
6	2	Laptops
7	3	Monitors
8	4	Pen
9	7	Papers
10	10	Pen

```
SELECT product_id, product_name
FROM left_products
UNION ALL
SELECT product_id, product_name
FROM right_products;
```

But what if we want duplicates?

Adding the ALL keyword to UNION will show duplicate records. i.e. Computers with ID 1, laptops with ID 2, Monitors with ID 3.

```
SELECT
first_name,
last_name
FROM directors
UNION
SELECT
first_name,
last_name
FROM actors
ORDER BY first_name;
```

	first_name character varying (150)	last_name character varying (150)
1	Darlan	Cunha
2	John	Travolta
3	Rafe	Spall
4	Irfan	Khan
5	Chen	Chang

I can use ORDER BY with UNION.

```
SELECT
first_name,
last_name,
'directors' AS "tablename"
FROM directors
UNION
SELECT
first_name,
last_name,
'actors' AS "tablename"
FROM actors
ORDER BY first_name;
```

	first_name character varying (150)	last_name character varying (150)	tablename text
41	Fernando	Meirelles	directors
42	Florian	Henckel von Donnersmarck	directors
43	Francis	Ford Coppola	directors
44	Francis	McDormand	actors
45	Gary	Oldman	actors
46	Gemma	Chan	actors
47	George	Lucas	directors

I can alias an extra column to show which table each record is coming from.


```
SELECT
first_name,
last_name,
'directors' AS "tablename"
FROM directors
UNION
SELECT
gender,
last_name,
'actors' AS "tablename"
FROM actors
ORDER BY first_name;
```

	first_name character varying 	last_name character varying (150) 	tablename text 
1	Ang	Lee	directors
2	Bruce	Lee	directors
3	Chan-wook	Park	directors
4	F	Akerman	actors
5	F	Andrews	actors
6	F	Baquero	actors

This query works because 'actors.gender' is the same data type as 'directors.first_name'

```
SELECT
first_name,
last_name,
'directors' AS "tablename"
FROM directors
UNION
SELECT
date_of_birth,
last_name,
'actors' AS "tablename"
FROM actors
ORDER BY first_name;
```

This query fails because 'actors.first_name' is type varchar and 'directors.date_of_birth' is of data type date.

ERROR: UNION types character varying and date cannot be matched
 LINE 8: date_of_birth,

SQL state: 42804 Character: 87

15.2 Union with Filters and Conditions

```
SELECT
first_name,
last_name,
nationality AS "filter"
FROM directors
WHERE nationality IN('American', 'Chinese', 'Japanese')
UNION
SELECT
first_name,
last_name,
gender AS "filter"
FROM actors
WHERE gender = 'F';
```

Combine all American, Chinese and Japanese Directors with all Female actors.

	first_name character varying (150)	last_name character varying (150)	filter character varying
1	Shelley	Duvall	F
2	Hayao	Miyazaki	Japanese
3	Faye	Wong	F
4	Connie	Nielsen	F
5	Jodie	Foster	F
6	Dandan	Song	F
7	Sam	Reimi	American

```
SELECT
first_name,
last_name,
date_of_birth AS "dob",
'directors' AS "tablename"
FROM directors
WHERE date_of_birth > '1990-12-31'
UNION
SELECT
first_name,
last_name,
date_of_birth AS "dob",
'actors' AS "tablename"
FROM actors
WHERE date_of_birth > '1990-12-31';
```

	first_name character varying (150) 🔒	last_name character varying (150) 🔒	dob date 🔒	tablename text 🔒
1	Hiroki	Doi	1999-08-10	actors
2	Dillon	Freasier	1996-03-06	actors
3	james	David	2010-01-01	directors
4	Suraj	Sharma	1993-03-21	actors
5	Yasmin	Paige	1991-06-24	actors
6	Kate	Hodgson	1995-06-08	actors

Get the first name and last name of all directors and actors born after 1990.

```

SELECT
first_name,
last_name,
'directors' AS "tablename"
FROM directors
WHERE LEFT(first_name,1) = 'A'
UNION
SELECT
first_name,
last_name,
'actors' AS "tablename"
FROM actors
WHERE first_name LIKE 'A%'

```

	first_name character varying (150) 🔒	last_name character varying (150) 🔒	tablename 🔒
1	Abbie	Cornish	actors
2	Adrien	Brody	actors
3	Aki	Maeda	actors
4	Alec	Guinness	actors
5	Alexandre	Rodrigues	actors
6	Alfred	Molina	actors
7	Alice	Bragg	actors

Get the first name and last name of all directors and actors where their first name starts with 'A'.

15.3 Union Tables with Different Number of Columns

```
SELECT
first_name,
last_name,
date_of_birth,
'directors' AS "tablename"
FROM directors
WHERE nationality IN('American', 'Chinese', 'Japanese')
UNION
SELECT
first_name,
last_name,
'actors' AS "tablename"
FROM actors
WHERE gender = 'F';
ERROR: each UNION query must have the same number of columns
LINE 10: first_name,
        ^
SQL state: 42601 Character: 158
```

Can we combine a different number of columns from each query? – **NO & YES!!**

```
CREATE TABLE union_1 (
    col1 INT,
    col2 INT
);

CREATE TABLE union_2 (
    col3 INT
);
```

```
SELECT
col1,
col2
FROM union_1
UNION
SELECT
NULL AS col1,
col3
FROM union_2
```

col1	col2
integer	integer

I can create a 'dummy' column, NULL to pad out the table with less rows so that both tables have the same number of rows in the union query.

15.4 Intersect with Tables

The INTERSECT operator returns any rows that are available in both results sets.

To be able to use INTERSECT:

- The order and number of the columns in the select list of all queries must be the same.
- The data type must be compatible too.

```
SELECT
product_id,
product_name
FROM left_products
INTERSECT
SELECT
product_id,
product_name
FROM right_products
```

	product_id integer	product_name character varying (100)
1	2	Laptops
2	1	Computers
3	3	Monitors

These records are in the same two tables.

```
SELECT column1, column2
FROM table1
INTERSECT
SELECT column1, column2
FROM table2
```

Intersect does not return duplicate records.

```
SELECT
first_name,
last_name
FROM directors
INTERSECT
SELECT
first_name,
last_name
FROM actors;
```

	first_name character varying (150)	last_name character varying (150)
1	Bruce	Lee
2	Terry	Jones

Let's intersect first name, last name of directors and actors table.

15.5 Except with Tables

The EXCEPT operator returns the rows in the first query that do not appear in the second query.

To be able to use EXCEPT:

- The order and number of the columns in the select list of all queries must be the same.
- The data type must be compatible too.

```
SELECT column1, column2
FROM table1
EXCEPT
SELECT column1, column2
FROM table2
```

```
SELECT
product_id,
product_name
FROM left_products
EXCEPT
SELECT
product_id,
product_name
FROM right_products
```

	product_id integer	product_name character varying (100)
1	5	Mics

Mics appears in the first table but not in the second table.

All the directors first and last names that do not appear in the actors table

```
SELECT
first_name,
last_name
FROM directors
EXCEPT
SELECT
first_name,
last_name
FROM actors
ORDER BY first_name
```

```
SELECT
first_name,
last_name
FROM directors
EXCEPT
SELECT
first_name,
last_name
FROM actors
WHERE gender = 'F'
ORDER BY first_name
```

	first_name character varying (150)	last_name character varying (150)
1	Ang	Lee
2	Bruce	Lee
3	Chan-wook	Park
4	Fernando	Meirelles
5	Florian	Henckel von Donnersmarck

List all the directors first name and last name unless they have the same first name in female actors.

	first_name character varying (150)	last_name character varying (150)
1	Ang	Lee
2	Chan-wook	Park
3	Fernando	Meirelles
4	Florian	Henckel von Donnersmarck
5	Francis	Ford Coppola
6	George	Lucas
7	Guillermo	del Toro

16. PostgreSQL Schemas

[16.1 What is a Schema?](#)

[16.2 Schema Operations: Add,Alter,Delete](#)

[16.3 Schema](#)

[Hierarchy](#)

[16.4 Move a Table to a New Schema](#)

[16.5 Schema Search Path](#)

[16.6 Alter Schema Ownership](#)

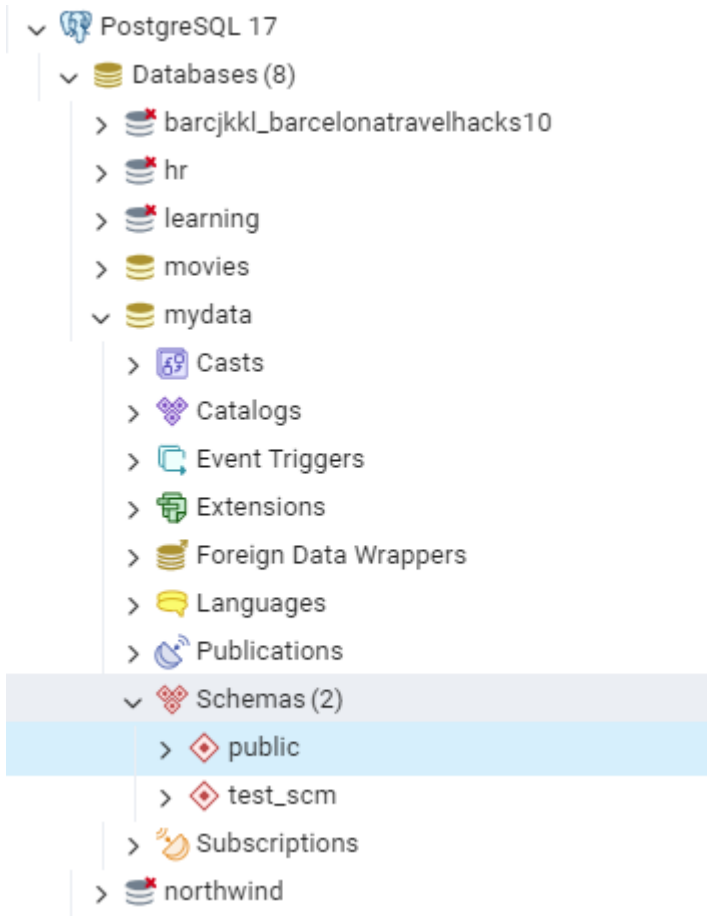
[16.7 Duplicate a Schema Along With All Data](#)

[16.8 What is a System Catalogue Schema?](#)

[16.9 Compare Tables & Columns in Two Schemas](#)

[16.10 Schemas and Privileges](#)

16.1 What is a Schema?



What is a Schema?

A schema is a namespace that contains named database objects such as: Tables, Views, Indexes, Data types, Functions, Stored procedures, Operators.

Each PostgreSQL Schema should be uniquely named and different from others.

What are the benefits of Schemas?

Schemas allow you to organise database objects, e.g. tables into more logical groups to make them more manageable.

Schemas allow multiple users to use one database without interfering with each other.

Schemas allow you to organise and limit database objects access to users.

PostgreSQL automatically creates a schema called public for every new database.

A database can contain one or multiple schemas and each schema belongs to only one database.

Two Schemas can have different objects with the same name.

16.2 Schema Operations: Add,Alter,Delete

CREATE SCHEMA schema_name will create schemas. Note how the database is called 'hr' and I have created a schema with the same name.

```
CREATE SCHEMA sales;  
CREATE SCHEMA hr;
```

Schema names must be UNIQUE within the database.

```
CREATE SCHEMA hr;  
ERROR: schema "hr" already exists  
SQL state: 42P06
```

ALTER SCHEMA schema_name
RENAME TO new_schema_name

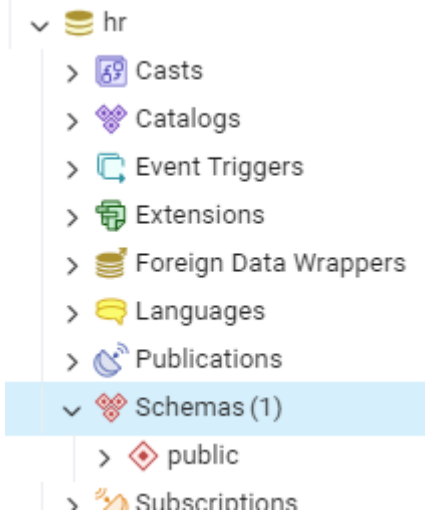
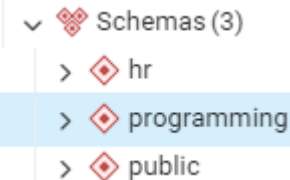
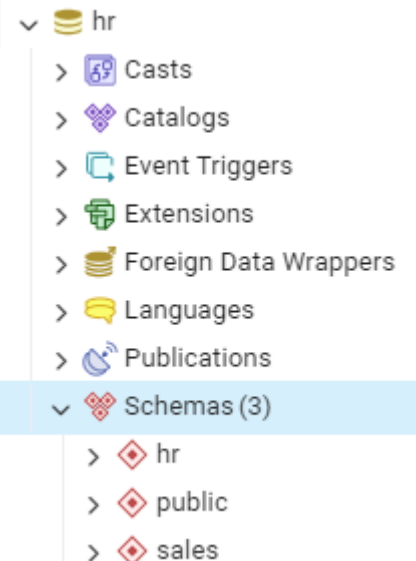
```
ALTER SCHEMA sales  
RENAME TO programming;
```

DROP SCHEMA schema_name

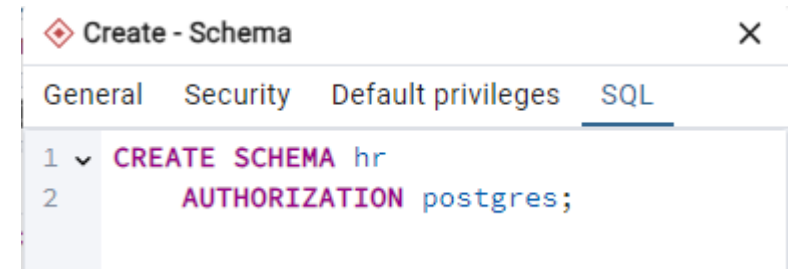
```
DROP SCHEMA programming;
```

```
DROP SCHEMA hr;
```

Dropping a schema is a destructive and non reversible process which will permanently erase all data.

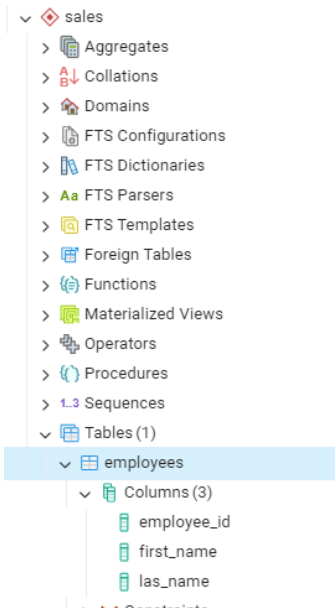
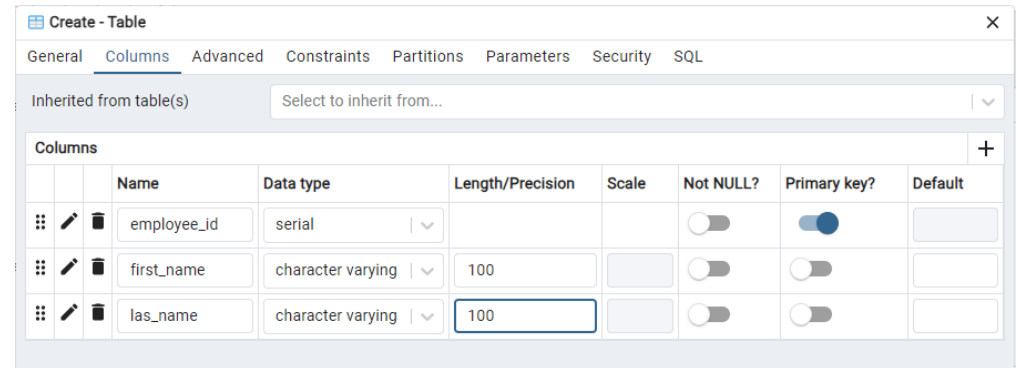


A schema can be created from PgAdmin 4 → Database → Schemas → right click → create schema. Give it a name and adjust parameters. The SQL tab will show the SQL that will be run to create this schema.

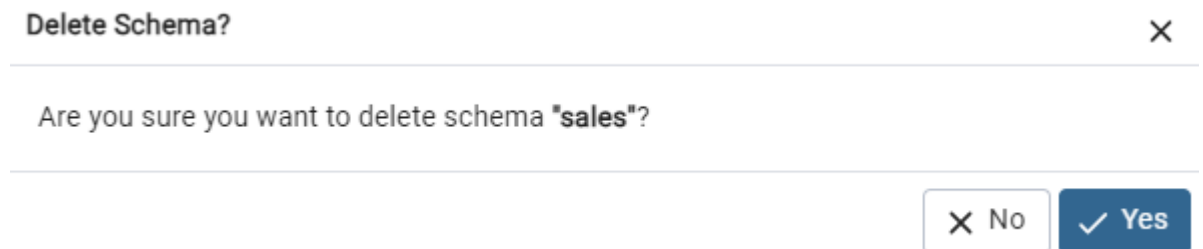


I have created two schemas with default parameters.

Right clicking on the sales schema → create table → called employees → with columns.



If I right click on the sales Schema → delete. I get a prompt.



This is because the schema contains data, i.e. the employees table and **PgAdmin tool will prompt us before we commit this destructive action** where as running from a terminal does not prompt before deleting schema.

16.3 Schema Hierarchy

Physical: host > cluster > database > schema > object name

Object access: database.schema.object_name eg hr.public.employees

```
SELECT * FROM hr.public.employees;
```

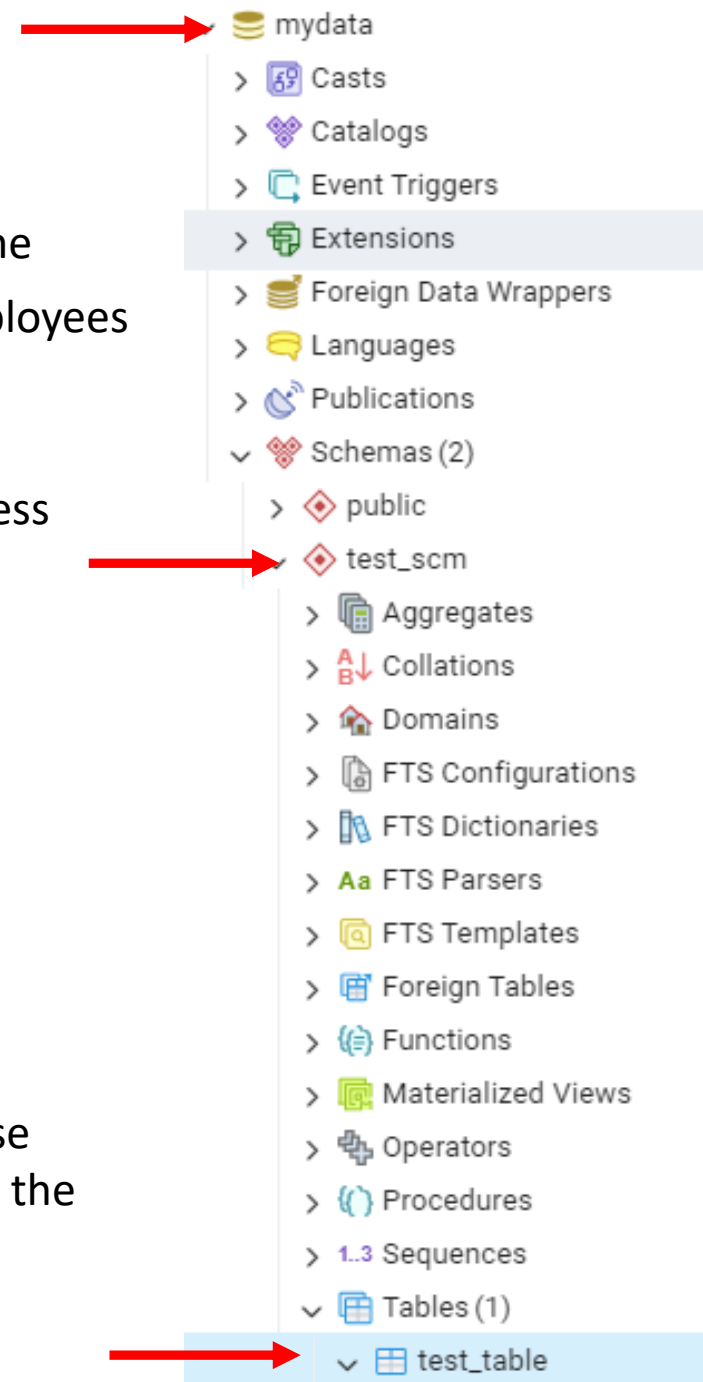
In a production database it will have many schemas and to access tables from the schemas, I need to specify the database_name.schema_name.table_name

```
SELECT * FROM employees;
```

For tables in the public schema I do not need to specify the full path to it but just this table name. This is because by default PostgreSQL search path maps to the public schema.

```
SELECT * FROM mydata.test_scm.test_table;
```

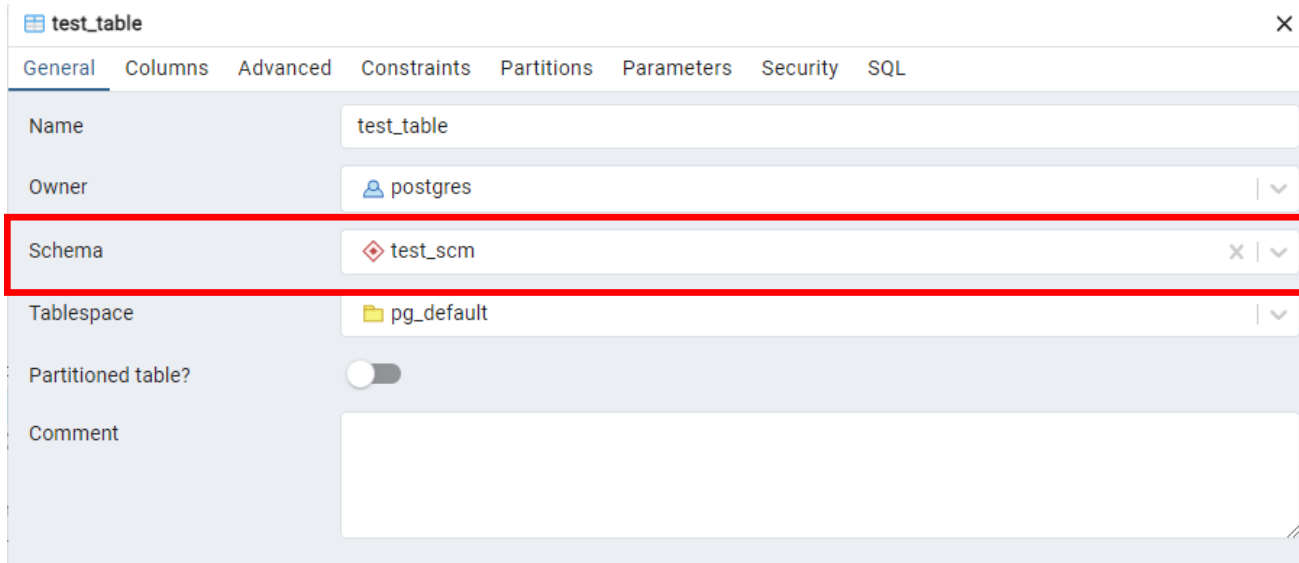
To access the 'test_table' in schema 'test_scm' within database 'mydata' I need to use the dot notation syntax above to specify the path to the table object.



16.4 Move a Table to a New Schema

ALTER TABLE table_name SET SCHEMA schema_name

```
ALTER TABLE test_scm.test_table SET SCHEMA public;
```



The screenshot shows the 'test_table' properties dialog in PgAdmin 4. The 'General' tab is selected. The 'Schema' field is highlighted with a red box and contains the value 'test_scm'. Other fields include 'Name' (test_table), 'Owner' (postgres), 'Tablespace' (pg_default), 'Partitioned table?' (toggle off), and 'Comment' (empty text area).

Property	Value
Name	test_table
Owner	postgres
Schema	test_scm
Tablespace	pg_default
Partitioned table?	☐
Comment	

I can also change a table schema in PgAdmin 4 by right clicking on the table → properties → general tab → schema → save

16.5 Schema Search Path

Normally we will refer to a table without its schema name e.g. test_table instead of a fully qualified name such as 'test_scm.test_table'.

When we reference a table using ITS NAME ONLY AND NOT FULLY QUALIFIED NAME, PostgreSQL searches for the table by using the SCHEMA SEARCH PATH.

SCHEMA SEARCH PATH is a list of schemas that is used to look for objects.

```
SELECT current_schema;
```

	current_schema name
1	public

```
SELECT * FROM test_table;  
ERROR: relation "test_table" does not exist  
LINE 1: SELECT * FROM test_table;  
                        ^  
SQL state: 42P01 Character: 15
```

My current schema is public so if I try and access a table from another schema without using the fully qualified name then it will fail. This is because our search path is public. What PostgreSQL is doing under the hood is add public to the table name i.e. 'public.table_name'.

```
SHOW search_path;
```

	search_path text
1	"\$user", public

The search path contains \$user is the user that is logged in. \$user is the default PostgreSQL user. The search_path to the schema. The search path is what is appended by default to all the queries that I run.

So the query 'SELECT * from test_table' would run 'SELECT * FROM postgresql.test_table' first and not find anything then it would run 'SELECT * FROM public.test_table'.

How to add a schema to the search path?

```
SET search_path TO test_scm, public;
```

```
SHOW search_path;
```

	search_path text
1	test_scm, public

I can add an additional search path using the SET search_path TO query, ensuring that I come separate all the schemas I want in the database search path and ensuring that I include public also.

```
SET search_path TO '$user', test_scm, public;
```

I can return \$user back to the search_path.

	search_path text
1	"\$user", test_scm, public

Now, because 'test_scm' is in the search path I can access it without needing the fully qualified name.

```
SELECT * FROM test_table;
```

test_id [PK] integer	test_name character varying (10)

However, PostgreSQL is first checking in postgresql, then test_scm to find the table data when a fully qualified name is not used so this can result in slower queries.

```
INSERT INTO test_table (test_name) VALUES ('testing_sc');  
Query returned successfully in 51 msec.
```

```
SET search_path TO test_scm, public, '$user';
```

```
SHOW search_path;
```

	search_path text
1	test_scm, public, "\$user"

```
INSERT INTO test_table (test_name) VALUES ('testing_sc');  
Query returned successfully in 48 msec.
```

The order of the search_paths matters!!!! Note how in the second insert query it is 3ms quicker because it went to look for 'test_table' first in the test_scm schema.

```
SET search_path TO test_scm, '$user';  
SHOW search_path;
```

	search_path	
	text	🔒
1	test_scm, "\$user"	

```
SELECT * FROM addresses;  
ERROR: relation "addresses" does not exist  
LINE 1: SELECT * FROM addresses;  
                        ^  
SQL state: 42P01 Character: 15
```

```
SET search_path TO test_scm, public, '$user';
```

If I do not add 'public' to the search_path then any tables under the public schema will not be reachable without the fully qualified name, i.e. 'public.table_name'

16.6 Alter Schema Ownership

```
ALTER SCHEMA hr OWNER TO elliot;
```

The owner of a schema can be changed by command or from PgAdmin 4.

The screenshot shows the 'hr' schema properties dialog in PgAdmin 4. The 'General' tab is selected. The 'Name' field contains 'hr'. The 'Owner' field is highlighted with a red rectangle and shows a user icon followed by 'elliott'. The 'Comment' field is empty.

hr	
General Security Default privileges SQL	
Name	hr
Owner	elliott
Comment	

16.7 Duplicate a Schema Along With All Data

```
CREATE DATABASE IF NOT EXISTS test_schema;

CREATE TABLE test_schema.public.songs (
    song_id SERIAL PRIMARY KEY,
    song_title VARCHAR(100)
);

INSERT INTO test_schema.public.songs (song_title) VALUES
('counting star'),
('Rolling On');
```

```
SELECT * FROM songs;
```

	song_id [PK] integer	song_title character varying (100)
1	1	counting star
2	2	Rolling On

Duplicate the Schema 'public' with all data.

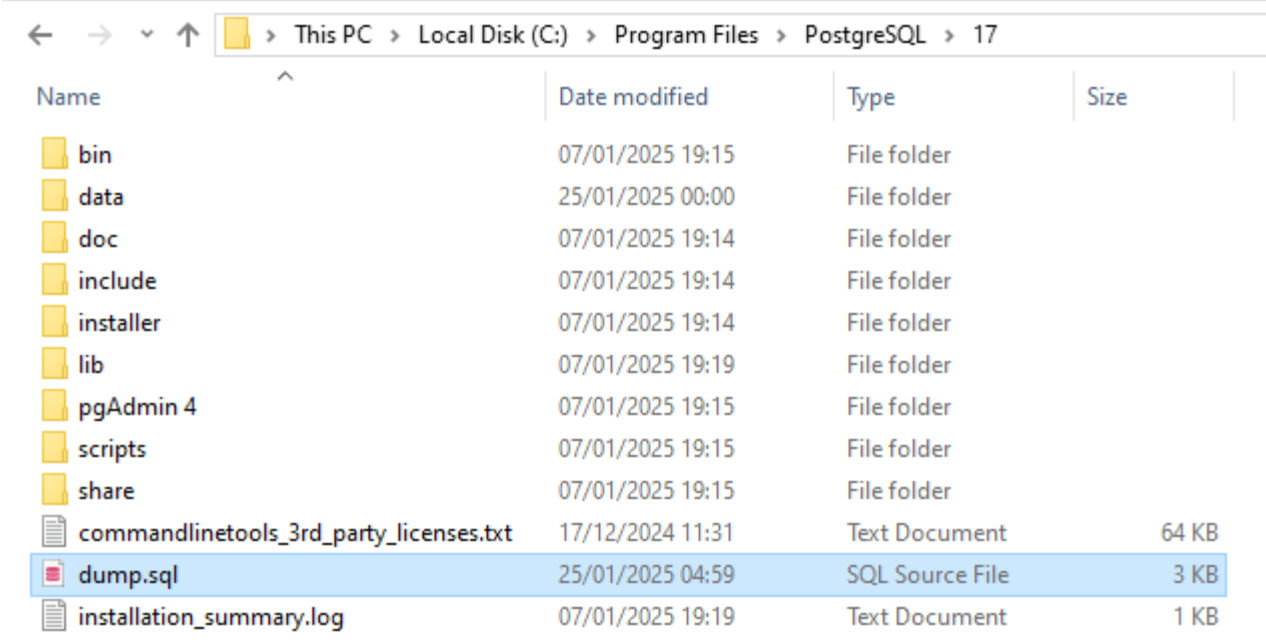
The first step is to make a dump of our PostgreSQL database 'public' schema using pg_dump. Open a command terminal as administrator.

Administrator: Command Prompt

```
C:\Program Files\PostgreSQL\17>
C:\Program Files\PostgreSQL\17>pg_dump -d test_schema -h localhost -U postgres -n public > dump.sql
Password:
C:\Program Files\PostgreSQL\17>
```

```
pg_dump -d test_schema -h localhost -U postgres -n public > dump.sql
```

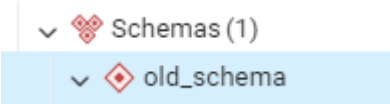

The output of this command is a SQL file of the whole database



This PC > Local Disk (C:) > Program Files > PostgreSQL > 17

Name	Date modified	Type	Size
bin	07/01/2025 19:15	File folder	
data	25/01/2025 00:00	File folder	
doc	07/01/2025 19:14	File folder	
include	07/01/2025 19:14	File folder	
installer	07/01/2025 19:14	File folder	
lib	07/01/2025 19:19	File folder	
pgAdmin 4	07/01/2025 19:15	File folder	
scripts	07/01/2025 19:15	File folder	
share	07/01/2025 19:15	File folder	
commandlinetools_3rd_party_licenses.txt	17/12/2024 11:31	Text Document	64 KB
dump.sql	25/01/2025 04:59	SQL Source File	3 KB
installation_summary.log	07/01/2025 19:19	Text Document	1 KB

Step 2: I renamed the 'public' schema to 'old_schema'



Schemas (1)

old_schema

```
Administrator: Command Prompt
C:\Program Files\PostgreSQL\17>psql -h localhost -U postgres -d test_schema -f dump.sql
Password for user postgres:
SET
SET
SET
SET
SET
SET
set_config
-----
(1 row)

SET
SET
SET
SET
CREATE SCHEMA
ALTER SCHEMA
COMMENT
SET
SET
CREATE TABLE
ALTER TABLE
CREATE SEQUENCE
ALTER SEQUENCE
ALTER SEQUENCE
ALTER TABLE
COPY 2
setval
-----
      2
(1 row)

ALTER TABLE

C:\Program Files\PostgreSQL\17>
```

Step 3: Import the previously dumped SQL file using:
psql -h localhost -U postgres -d test_schema -f dump.sql

Now there is a new schema called 'public' which has the 'songs' table and data.

```
SELECT * FROM public.songs;
```

	song_id [PK] integer	song_title character varying (100)
1	1	counting star
2	2	Rolling On

- ✓ Schemas (2)
 - > old_schema
 - ✓ public
 - > Aggregates
 - > Collations
 - > Domains
 - > FTS Configurations
 - > FTS Dictionaries
 - > FTS Parsers
 - > FTS Templates
 - > Foreign Tables
 - > Functions
 - > Materialized Views
 - > Operators
 - > Procedures
 - > 1.3 Sequences
 - ✓ Tables (1)
 - ✓ songs
 - > Columns

16.8 What is a System Catalogue Schema?

In addition to public and user-created schemas, each database contains a 'pg_catalog' schema. PostgreSQL stores the metadata information about the database and cluster in the schema 'pg_catalog'. This information is partially used by PostgreSQL itself to keep track of things itself, but it also is resented so that external people / processes can understand the inside of the databases too.

'pg_catalog' schema contains system tables and all the built-in data types, functions and operators.

'pg_catalog' is always effectively part of the search path. Although if it is not named explicitly in the path, then it is **implicitly searched before searching the path's schemes**.

```
SHOW search_path;
```

	search_path text
1	"\$user", public

```
Pg_catalog,$user,public
```

However, you can explicitly place 'pg_catalog' at the end of your search path if you prefer to have user-defined names override built-in- names.

```
SET search_path TO '$user',public,pg_catalog;
```

In 'pg_catalog' all system table names begin with 'pg_' so it is best to **avoid such names in the user defined tables** to ensure that the database does not suffer a conflict.

How to view 'pg_catalog' or in fact all schemas including system.

```
SELECT * FROM information_schema.schemata;
```

	catalog_name name	schema_name name	schema_owner name	default_character_set_catalog name	default_character_set_schema name	default_character_set_name name	sql_path character varying
1	test_schema	public	pg_database_owner	[null]	[null]	[null]	[null]
2	test_schema	old_schema	pg_database_owner	[null]	[null]	[null]	[null]
3	test_schema	information_schema	postgres	[null]	[null]	[null]	[null]
4	test_schema	pg_catalog	postgres	[null]	[null]	[null]	[null]
5	test_schema	pg_toast	postgres	[null]	[null]	[null]	[null]

The PostgreSQL catalog has a pretty solid rule: **look but don't touch or alter!**

While PostgreSQL stores all this information in tables like any other application would, the data in the tables is fully managed by PostgreSQL itself, and should not be modified unless an absolute emergency, and even then a rebuild is likely in order afterwards.

16.9 Compare Tables & Columns in Two Schemas

The PostgreSQL query below compares column names in tables between two PostgreSQL schemas. It shows missing columns in either of the schemas.

```
SELECT
COALESCE(c1.table_name, c2.table_name) AS table_name,
COALESCE(c1.column_name, c2.column_name) AS table_column,
c1.column_name AS schema1,
c2.column_name AS schema2
FROM
(
    SELECT table_name, column_name
    FROM information_schema.columns c
    WHERE c.table_schema = 'schema_1'
) c1
FULL JOIN
(
    SELECT table_name, column_name
    FROM information_schema.columns c
    WHERE c.table_schema = 'schema_2'
) c2
ON c1.table_name = c2.table_name AND c1.column_name = c2.column_name
WHERE c1.column_name IS NULL OR c2.column_name IS NULL
ORDER BY table_name, table_column;
```

```

SELECT
COALESCE(c1.table_name, c2.table_name) AS table_name,
COALESCE(c1.column_name, c2.column_name) AS table_column,
c1.column_name AS schema1,
c2.column_name AS schema2
FROM
(
    SELECT table_name, column_name
    FROM information_schema.columns c
    WHERE c.table_schema = 'hr'
) c1
FULL JOIN
(
    SELECT table_name, column_name
    FROM information_schema.columns c
    WHERE c.table_schema = 'public'
) c2
ON c1.table_name = c2.table_name AND c1.column_name = c2.column_name
WHERE c1.column_name IS NULL OR c2.column_name IS NULL
ORDER BY table_name, table_column;

```

	table_name 	table_column 	schema1 	schema2 
1	countries	country_id	[null]	country_id
2	countries	country_name	[null]	country_name
3	countries	region_id	[null]	region_id
4	departments	department_id	[null]	department_id
5	departments	department_name	[null]	department_name
6	departments	location_id	[null]	location_id
7	departments	manager_id	[null]	manager_id

16.10 Schemas and Privileges

Users can only access objects in the schemas that they own.

Two schema access level rights

*USAGE – to access schema

*CREATE – to create objects like tables etc in a schema

USAGE allows users to access the objects in the schema that they do not own, we must grant the usage privilege of the schema

GRANT USAGE ON SCHEMA schema_name TO role_name

I have created a schema in the 'hr' database called 'private' and set the **owner to postgres**. T1 is in the private schema.

If I log into the database as user 'Elliott' from a terminal and try and view data from table 't1' I get a permission denied message for the **SCHEMA**.

```
Server [localhost]:
Database [postgres]:
Port [5432]:
Username [postgres]: elliot
Password for user elliot:
psql (17.2)
WARNING: Console code page (850) differs from Windows code page (1252)
        8-bit characters might not work correctly. See psql reference
        page "Notes for Windows users" for details.
Type "help" for help.

postgres=> \c hr
You are now connected to database "hr" as user "elliot".
hr=> \d
               List of relations
 Schema |   Name   | Type  | Owner
-----+-----+-----+-----
 public | countries | table | postgres
 public | departments | table | postgres
 public | employees | table | postgres
 public | job_history | table | postgres
 public | jobs      | table | postgres
 public | locations | table | postgres
 public | regions   | table | postgres
(7 rows)

hr=> SELECT * FROM private.t1;
ERROR:  permission denied for schema private
```

```
GRANT USAGE ON SCHEMA private TO elliot;
```

```
hr=> SELECT * FROM private.t1;  
ERROR: permission denied for table t1
```

After granting 'Elliott' USAGE privileges to private schema, Elliott can see the schema but does not have privileges for table 't1'

```
GRANT SELECT ON ALL TABLES IN SCHEMA private TO elliot;
```

```
hr=> SELECT * FROM private.t1;  
id  
----  
(0 rows)
```

After granting 'Elliott' SELECT privileges on all tables in the 'private' schema, this user can now run select queries.

```
hr=> CREATE TABLE private.t2 (id INT);  
ERROR: permission denied for schema private  
LINE 1: CREATE TABLE private.t2 (id INT);  
          ^  
hr=>
```

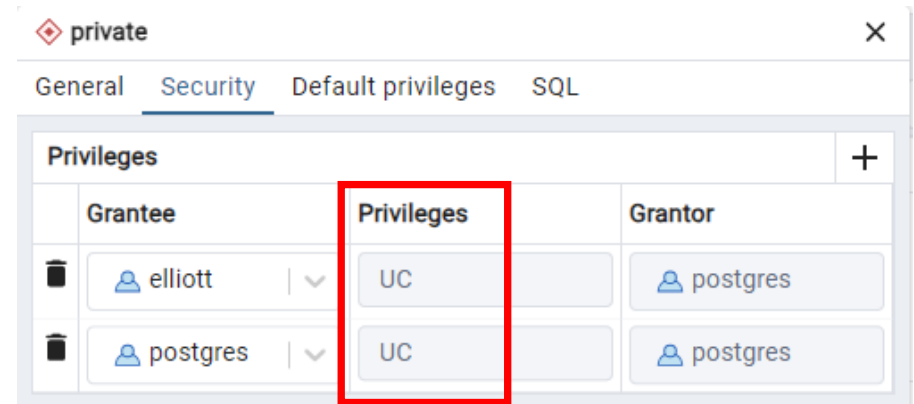
But 'Elliott' has USAGE privileges only for the 'private' schema so creating tables is not allowed.

```
GRANT CREATE ON SCHEMA private TO elliot;
```

Now I see that postgres and Elliott both have CREATE & USAGE privileges on the 'private' schema,

```
hr=> CREATE TABLE private.t2 (id INT);  
CREATE TABLE
```

Grantee, 'Elliott' can now create tables.



private			
General Security Default privileges SQL			
Privileges +			
	Grantee	Privileges	Grantor
	elliott	UC	postgres
	postgres	UC	postgres

By default all postgres and all users have USAGE and CREATE privileges on public schemas so plan accordingly for security needs.

17. Exploring Array Functions

[17.1 Constructing Arrays & Ranges](#)

[17.2 Using Comparison operators](#)

[17.3 Using Inclusion operators](#)

[17.4 Array Constructions](#)

[17.5 Array Metadata Functions](#)

[17.6 Array Search Functions](#)

[17.7 Array Modification Functions](#)

[17.8 Array Comparison with
IN,ALL,ANY & SOME](#)

[17.9 Formatting & Converting an
Array](#)

[17.10 Using Arrays in Tables](#)

[17.11 Insert Data into Arrays](#)

[17.12 Query Array Data](#)

[17.13 Modify Array Data](#)

[17.14 Array Dimensions
are Ignored!](#)

[17.15 Display All Array Elements](#)

[17.16 Using Multi-dimensional
Arrays](#)

[17.17 Array vs JSONB](#)

[17.18 Single & Multi-
dimensional Arrays](#)

17.1 Constructing Arrays & Ranges

An array is an ordered collection of elements.

The 'Cardinality' of an array is the number of elements in the array. This value can be zero or more.

Individual elements can be accessed from an array by enclosing their 'subscripts' or position in square brackets. For example `customers[2]` would refer to the second element of the `customers` array.

```
range_type(lower_bound, upper_bound)
```

```
range_type: INT4RANGE, NUMRANGE, TSRANGE, TSTZRANGE, DATERANGE
```

```
range_type(lower_bound, upper_bound, open_close)
```

```
open_close: '[' - Closed_closed
```

```
            '[' - closed_open
```

```
            '(' - open_closed
```

```
            '(' - open_open
```

Square brackets indicate the range is close

Round bracket indicates the range is open

```
SELECT
INT4RANGE(1,6) AS "default [( = closed - opened",
NUMRANGE(1.4213, 62986, '[') AS "[] closed - closed",
DATERANGE('20100101', '20201220', '()') AS "dates () = opened - opened",
TSRANGE(LOCALTIMESTAMP, LOCALTIMESTAMP + INTERVAL '8 days', '[') AS "opened-closed";
```

	default [(= closed - opened int4range	[] closed - closed numrange	dates () = opened - opened daterange	opened-closed tsrange
1	[1,6)	[1.4213,62986]	[2010-01-02,2020-12-20)	("2025-01-28 09:27:26.405801","2025-02-05 09:27:26.405801"]

The 1st selection is for a closed – open range, seen by the square then round brackets.

The 2nd selection is for a closed closed range which is indicated by square brackets on both sides.

The 3rd selection is for a both sides open range, seen by both round brackets. Note how it has converted the dates into a range by ordering the lower date as the lower_bound and the higher date as the upper_bound.

The 4th selection is using the localtimestamp as the lower_bound and the upper_bound is the current timestamp plus 8 days. The brackets are round then square as described.

```
SELECT
ARRAY[1,2,3] AS "INT arrays",
ARRAY[2.12225::FLOAT] AS "floating numbers with explicitly
typing",
ARRAY[CURRENT_DATE, CURRENT_DATE +5]
```

	INT arrays integer[]	floating numbers with explicitly typing double precision[]	array date[]
1	{1,2,3}	{2.12225}	{2025-01-28,2025-02-02}

Array [value1, value2, value3, value4] – If you do no specify the type of the value1, then PostGreSQL will try and evaluate the type itself.

17.2 Using Comparison operators

```
SELECT
ARRAY[1,2,3,4] = ARRAY[1,2,3,4] AS "Equality",
ARRAY[1,2,3,4] = ARRAY[2,3,4] AS "Equality",
ARRAY[1,2,3,4] <> ARRAY[2,3,4,5] AS "Not Equal to",
ARRAY[1,2,3,4] < ARRAY[2,3,4,5] AS "Less than",
ARRAY[1,2,3,4] <= ARRAY[2,3,4,5] AS "Less than and equal to",
ARRAY[1,2,3,4] > ARRAY[2,3,4,5] AS "Greater than",
ARRAY[1,2,3,4] >= ARRAY[2,3,4,5] AS "Greater than and equal to";
```

	Equality boolean	Equality boolean	Not Equal to boolean	Less than boolean	Less than and equal to boolean	Greater than boolean	Greater than and equal to boolean
1	true	false	true	true	true	false	false

```
SELECT
INT4RANGE(1,4) @> INT4RANGE(2,3) AS "Contains",
DATERANGE(CURRENT_DATE, CURRENT_DATE + 30) @> CURRENT_DATE + 15 AS "Contains value",
NUMRANGE(1.6, 5.2) && NUMRANGE(0,4);
```

	Contains boolean	Contains value boolean	?column? boolean
1	true	true	true

I can compare ranges where I see if the second range is contained within the first range (@>). I am using the && operator to see if the second range overlaps the first range. In this case a range of 0 to 4 overlaps a range of 1.6 to 5.2.

17.3 Using Inclusion operators

Inclusion operators @>, <@, && can be used to check if a range or array is in another range or array, i.e. array b is contained in array a or do they overlap.

```
SELECT
ARRAY[1,2,3,4] @> ARRAY[2,3,4] AS "contains",
ARRAY['a','b'] <@ ARRAY['a','b'] AS "contained by",
ARRAY[1,2,3,4] && ARRAY[2,3,4] AS "overlap";
```

	contains boolean	contained by boolean	overlap boolean
1	true	true	true

1. Is the right hand array contained within the left hand array? True
2. Is the left hand array contained by the right hand array? True
3. Does the right hand array overlap with the left hand array? True

```
SELECT
INT4RANGE(1,4) @> INT4RANGE(2,3) AS "Contains",
DATERANGE(CURRENT_DATE, CURRENT_DATE + 30) @> CURRENT_DATE + 15 AS "Contains value",
NUMRANGE(1.6, 5.2) && NUMRANGE(0,4) AS "overlap";
```

	Contains boolean	Contains value boolean	overlap boolean
1	true	true	true

1. Is the right hand range contained within the left hand range? True
2. Is the right hand range contained within the left hand range? True
3. Does the right hand range overlap with the left hand range? True

17.4 Array Constructions

How to construct arrays with concatenation operator || and built in array constructors

```
SELECT
ARRAY[1,2,3] || ARRAY[4,5,6] AS "combine arrays",
ARRAY_CAT(ARRAY[1,2,3], ARRAY[4,5,6]) AS "Combine arrays via ARRAY_CAT";
```

	combine arrays integer[]	Combine arrays via ARRAY_CAT integer[]
1	{1,2,3,4,5,6}	{1,2,3,4,5,6}

Combining an array with:

1. ||
2. ARRAY_CAT function

Note the output is an integer. It is important to ensure that elements of both arrays are of the same data type.

```
SELECT
4 || ARRAY[1,2,3] AS "adding to an array",
ARRAY_PREPEND(4, ARRAY[1,2,3]) AS "using prepend";
```

	adding to an array integer[]	using prepend integer[]
1	{4,1,2,3}	{4,1,2,3}

Prepending an array with:

1. ||
2. ARRAY_PREPEND function

```
SELECT
ARRAY[1,2,3] || 4 AS "adding to end of an array",
ARRAY_APPEND(ARRAY[1,2,3], 4) AS "using append";
```

	adding to end of an array integer[]	using append integer[]
1	{1,2,3,4}	{1,2,3,4}

appending an array with:

1. ||
2. ARRAY_APPEND function

17.5 Array Metadata Functions

ARRAY_NDIMS(anyarray) – dimensions in an array. Return is always an integer

```
SELECT  
ARRAY_NDIMS(ARRAY[[1],[2]]) AS "dimensions",  
ARRAY_NDIMS(ARRAY[[1,2,3], [4,5,6]]) AS "dimensions";
```

	dimensions integer	dimensions integer
1	2	2

ARRAY_DIMS(anyarray) – returns a TEXT representation of an array's dimensions.

```
SELECT  
ARRAY_DIMS(ARRAY[[1],[2]]) AS "dimensions",  
ARRAY_DIMS(ARRAY[[1,2,3], [4,5,6]]) AS "dimensions";
```

	dimensions text	dimensions text
1	[1:2][1:1]	[1:2][1:3]

ARRAY_LENGTH(anyarray, int) – returns the length of the requested array dimension as an integer.

```
SELECT  
ARRAY_LENGTH(array[1,2,3,4], 1) AS "array_length",  
ARRAY_LENGTH(array[1,2,3,4,5,6], 1) AS "array_length";
```

	array_length integer	array_length integer
1	4	6

```
SELECT ARRAY_LENGTH(ARRAY[],1);  
ERROR: cannot determine type of empty array  
LINE 1: SELECT ARRAY_LENGTH(ARRAY[],1)  
                        ^  
HINT: Explicitly cast to the desired type, for example ARRAY[]::integer[].  
SQL state: 42P18 Character: 21
```

```
SELECT ARRAY_LENGTH(ARRAY[]::INTEGER[],1);
```

	array_length integer
1	[null]

ARRAY_LOWER(anyarray, int) – returns lower bound of the requested array dimension

```
SELECT  
ARRAY_LOWER(array[1,2,3,4], 1) AS "array_lower",  
ARRAY_LOWER(array[1,4], 1) AS "array_lower";
```

	array_lower integer	array_lower integer
1	1	1

ARRAY_UPPER(anyarray, int) – returns upper bound of the requested array dimension

```
SELECT  
ARRAY_UPPER(array[1,2,3,4], 1) AS "array_upper",  
ARRAY_UPPER(array[1,4], 1) AS "array_upper";
```

	array_upper integer	array_upper integer
1	4	2

Note that ARRAY_LOWER and ARRAY_UPPER do not return lowest and highest values in the array but the upper bounds of the number of elements, i.e. the dimension of the array.

CARDINALITY(anyarray) – returns the cardinality of the array dimension or the total numbers of elements in an array

```
SELECT  
CARDINALITY(ARRAY[[1],[2],[3],[4]]) AS "cardinality",  
CARDINALITY(ARRAY[[1],[2],[3],[4],[5]]) AS "cardinality";
```

	cardinality integer	cardinality integer
1	4	5

Find more array functions in PostgreSQL documentation:

<https://www.postgresql.org/docs/9.1/functions-array.html>

17.6 Array Search Functions

`ARRAY_POSITION(anyarray, anyelement)` – returns the subscript of the first occurrence of the second argument in the array, starting at the element indicated by the third argument or at the first element

```
SELECT  
ARRAY_POSITION(ARRAY['Jan','Feb','Mar'],'Feb') AS "position of feb",  
ARRAY_POSITION(ARRAY[1,2,3,4],3) AS "position of 3";
```

	position of feb integer	position of 3 integer
1	2	3

Note that in most traditional programming languages, PHP, python, JS etc, an array is zero indexed i.e the first element is position zero, the second element position 1 etc. In PostgreSQL the arrays positions start from 1.

```
SELECT ARRAY_POSITION(ARRAY[1,2,2,3,4],2) AS "position of 2";
```

	position of 2 integer
1	2

`Array_position` will find the FIRST occurrence only, even if there are more occurrences

```
SELECT ARRAY_POSITION(ARRAY[1,2,2,3,4],2, 3) AS "position of 2";
```

	position of 2 integer
1	3

The third argument is where I would like it to start searching. In this case I say that I want it to start looking from position 3 of the array and it returns that the number 2 is in the 3rd position.

```
SELECT ARRAY_POSITIONS(ARRAY[1,2,2,3,4],2) AS "position of 2";
```

	position of 2 integer[]
1	{2,3}

`Array_positions` will find ALL occurrences of what we are searching for in an array. In this case the value '2' is in the array in the 2nd and third position.

17.7 Array Modification Functions

`ARRAY_REMOVE(anyarray, anyelement)` – is used to remove ALL elements equal to the given value from the array (array must be one-dimensional).

```
SELECT  
ARRAY_REMOVE(ARRAY[1,2,3,4],4) AS "array_remove",  
ARRAY_REMOVE(ARRAY[1,2,3,4,4,4,4],4) AS "array_remove";
```

	array_remove integer[]	array_remove integer[]
1	{1,2,3}	{1,2,3}

`ARRAY_REPLACE(anyarray, fromelement, toelement)` – is used to replace ALL elements equal to the given value from the array where the second argument is the from value you would like to replace and the third argument is the to value that you would like to replace. (array must be one-dimensional).

```
SELECT  
ARRAY_REPLACE(ARRAY[1,2,3,4],2,10) AS "array_replace",  
ARRAY_REPLACE(ARRAY[1,2,3,4,4],4,10) AS "array_replace";
```

	array_replace integer[]	array_replace integer[]
1	{1,10,3,4}	{1,2,3,10,10}

17.8 Array Comparison with IN,ALL,ANY & SOME

Expression IN (value [, ...])

The right hand side is a parenthesised list of scalar expressions.

The result is true if the left hand expression's result is equal to ANY of the right hand expressions.

```
SELECT  
20 IN (1,2,20,40) AS "result",  
25 IN (1,2,20,40) AS "no result";
```

	result boolean	no result boolean
1	true	false

The long hand way of writing IN would be:

Expression = value1 OR Expression = value2 OR expression = value3 etc...

Expression NOT IN (value [, ...]).

The right hand side is a parenthesised list of scalar expressions.

The result is true if the left hand expression's result is not equal to ANY of the right hand expressions.

```
SELECT  
20 NOT IN (1,2,20,40) AS "no result",  
25 NOT IN (1,2,20,40) AS "result";
```

	no result boolean	result boolean
1	false	true

The long hand way of writing NOT IN would be:

Expression <> value1 AND Expression <> value2 AND expression <> value3 etc...

Expression ALL (array expression)

The right hand side is a parenthesised expression, which must yield an array value.

The left hand expression is evaluated and compared to each element in the array using the given operator, which must yield a Boolean result.

The result of ALL is “true” if all comparisons yield true (including the case where the array has zero elements). The result is “false” if false result is found.

```
SELECT
25 = ALL (ARRAY[20,25,30,35,40]) AS "no result",
25 = ALL (ARRAY[25,25]) AS "result";
```

	no result boolean	result boolean
1	false	true

The right hand side is a parenthesised expression, which must yield an array value.

The left hand expression is evaluated and compared to each element in the array using the given operator, which must yield a Boolean result.

The result of ANY is “true” if any comparisons yield true

The result is “false” if false if no true result is found. (including the case where the array has zero elements).

```
SELECT
25 = ANY(ARRAY[20,25,30,35,40]) AS "result",
20 = ANY(ARRAY[1,2,3]) AS "no result",
25 = ANY(ARRAY[1,2,25,NULL]) AS "= ANY with NULLS",
25 <> ANY(ARRAY[1,2,25,NULL]) AS "!= ANY with NULLS",
25 = SOME(ARRAY[1,2,3,4]) AS "SOME",
20 = SOME (ARRAY[1,2,3,4,25]) AS "SOME results";
```

Some and any perform the same function.

	result boolean	no result boolean	= ANY with NULLS boolean	!= ANY with NULLS boolean	SOME boolean	SOME results boolean
1	true	false	true	true	false	false

17.9 Formatting & Converting an Array

STRING_TO_ARRAY(input_text, delimiter_text [, opt null string])

Function is used to convert a string into an array elements using supplied delimiter and optional null string.

```
SELECT  
STRING_TO_ARRAY('1,2,3,4', ',') AS "without NULL string",  
STRING_TO_ARRAY('1,2,3,4,abc', ',','abc') AS "with NULL string";
```

	without NULL string text[]	with NULL string text[]
1	{1,2,3,4}	{1,2,3,4,NULL}

Note how we can specify any text that we want to convert to a null value

```
SELECT STRING_TO_ARRAY('1,2,3,4,,6', ',','') AS "empty values as NULLS";
```

	empty values as NULLS text[]
1	{1,2,3,4,NULL,6}

Here I can convert empty values to NULLS

ARRAY_TO_STRING(array, delimiter_text [, opt null string])

Function is used to split an array into string using supplied delimiter and optional null string

```
SELECT  
ARRAY_TO_STRING(ARRAY[1,2,NULL,4], '|', 'EMPTY_DATA') AS "array_to_string with NULL",  
ARRAY_TO_STRING(ARRAY[1,2,3,4], '$') AS "array_to_string";
```

	array_to_string with NULL text	array_to_string text
1	1 2 EMPTY_DATA 4	1\$2\$3\$4

17.10 Using Arrays in Tables

PostgreSQL allows the definition of a column to be an array of any valid data type including built-in-type, user-defined-type or enumerated type.

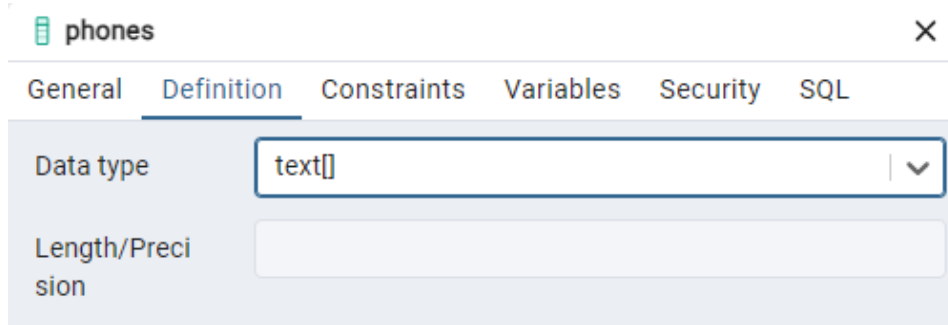
Every data type has its own companion array type. I.e. integer has an integer[] array type, character has character[] array type etc...

Basically add brackets [] to the base data type to make it an array e.g. name VARCHAR(100) []

```
CREATE TABLE teachers (  
    teacher_id SERIAL PRIMARY KEY,  
    name VARCHAR(150),  
    phones TEXT []  
);
```

I can define a column as an array using the square brackets or the ARRAY keyword.

```
CREATE TABLE teachers1 (  
    teacher_id SERIAL PRIMARY KEY,  
    name VARCHAR(150),  
    phones TEXT ARRAY  
);
```



The screenshot shows a dialog box for defining a column named 'phones'. The 'Definition' tab is selected. The 'Data type' dropdown is set to 'text[]'. The 'Length/Precision' field is empty.

Note how the data type is a text array denoted by text[].

The phones column is a one dimensional array that holds various phone numbers that a contact may have.

17.11 Insert Data into Arrays

We use single quote to wrap the array or use ARRAY function – ARRAY[value1, value2]

For non-text data, use curly braces – '{value1, value2}'

For text-data use double quotes when using curly braces – '{"text1","text2"}'

And for text in an array function use single quotes – ARRAY['text1','text2']

```
INSERT INTO teachers (name, phones) VALUES
('Linda', '{"(111)-123-4567"}'),
('Jeff', '{"(222)-555-9999","(444)-786-12345"}');

INSERT INTO teachers (name, phones) VALUES
('Adam', ARRAY['(111)-222-3333','(555)-666-7777']);
```

17.12 Query Array Data

Notice the phones column is an array and each row contains an array (denoted by curly brackets)

```
SELECT * FROM teachers;
```

	teacher_id [PK] integer	name character varying (150)	phones text[]
1	1	Linda	{{(111)-123-4567}}
2	2	Jeff	{{(222)-555-9999,(444)-786-12345}}
3	3	Adam	{{(111)-222-3333,(555)-666-7777}}

```
SELECT name, phones[1] AS "phone1", phones[2] AS "phone2" FROM teachers;
```

I can get the first element of the array denoting it's position (1st, 2nd etc..) in the array within square brackets. Note that if there is no value in an array position it returns null.

	name character varying (150)	phone1 text	phone2 text
1	Linda	(111)-123-4567	[null]
2	Jeff	(222)-555-9999	(444)-786-12345
3	Adam	(111)-222-3333	(555)-666-7777

```
SELECT name, phones[1]
FROM teachers
WHERE phones[1]; = '(111)-222-3333';
```

	name character varying (150)	phones text
1	Adam	(111)-222-3333

Search all rows to find the record where the first element in the array is equal to

```
SELECT *
FROM teachers
WHERE '(111)-222-3333' = ANY (phones);
```

	teacher_id [PK] integer	name character varying (150)	phones text[]
1	3	Adam	{{(111)-222-3333,(555)-666-7777}}

I can use any to get rows where a value is in an array.

17.13 Modify Array Data

```
UPDATE teachers
SET phones[2] = '(800)-123-456'
WHERE teacher_id = 1;
```

I can use square brackets to denote the element of the array that I want to update.

```
SELECT * FROM teachers;
```

	teacher_id [PK] integer	name character varying (150)	phones text[]
1	2	Jeff	{{(222)-555-9999,(444)-786-12345}}
2	3	Adam	{{(111)-222-3333,(555)-666-7777}}
3	1	Linda	{{(111)-123-4567,(800)-123-456}}

17.14 Array Dimensions are Ignored!

```
CREATE TABLE teachers2 (  
  teacher_id SERIAL PRIMARY KEY,  
  name VARCHAR(150),  
  phones TEXT ARRAY[1]  
);
```

We can introduce array dimensions but in the PostgreSQL world, dimensions are ignored.

Here I create a table where the phones column is an array but specified as an array limited to only one element.

```
INSERT INTO teachers2 (name, phones) VALUES  
( 'Adam', ARRAY[ '(111)-222-3333', '(555)-666-7777' ] );  
INSERT 0 1 Query returned successfully in 42 msec.
```

	teacher_id [PK] integer	name character varying (150)	phones text[]
1	1	Adam	{{(111)-222-3333,(555)-666-7777}}

Even though I have defined the column phones as an array of only one dimension, PostgreSQL still allows me to insert an array of two dimensions.

DIMENSIONS ARE IGNORED IN POSTGRESQL!

17.15 Display All Array Elements

UNNEST(anyArray) Function is used to expand an array to a set of rows.

Once expanded to a bunch of rows it works like any other SQL table and things like ORDER BY can be used on the data.

```
SELECT  
  teacher_id,  
  name,  
  UNNEST(phones)  
FROM teachers  
ORDER BY 3;
```

	teacher_id [PK] integer	name character varying (150)	unnest text
1	1	Linda	(111)-123-4567
2	3	Adam	(111)-222-3333
3	2	Jeff	(222)-555-9999
4	2	Jeff	(444)-786-12345
5	3	Adam	(555)-666-7777
6	1	Linda	(800)-123-456

17.16 Using Multi-dimensional Arrays

Lets create a table “students” where we will store student grades along with the year, i.e. multidimensional. Notice the [] [] to create a multi-dimensional array. Adding the second set of brackets tells PostGRES SQL that our data type is still an array but now our array of integers has two dimensions.

```
CREATE TABLE students (  
  student_id SERIAL PRIMARY KEY,  
  student_name VARCHAR(100),  
  student_grade INTEGER[][]  
);
```

```
INSERT INTO students (student_name, Student_grade) VALUES  
('s1', '{90,2020}');
```

```
SELECT * FROM  
students;
```

	student_id [PK] integer	student_name character varying (100)	student_grade integer[]
1	1	s1	{90,2020}

```
INSERT INTO students  
(student_name, Student_grade)  
VALUES  
('s2', '{80,2019}'),  
('s3', '{70,2019}'),  
('s4', '{60,2019}');
```

```
SELECT  
student_id,  
student_grade[1] AS "grade",  
student_grade[2] AS "year"  
FROM students;
```

	student_id [PK] integer	grade integer	year integer
1	1	90	2020
2	2	80	2019
3	3	70	2019
4	4	60	2019

```
SELECT * FROM students WHERE student_grade @> '{2020}';
```

```
SELECT * FROM students WHERE 2020 = ANY(student_grade);
```

	student_id [PK] integer	student_name character varying (100)	student_grade integer[]
1	1	s1	{90,2020}

```
SELECT * FROM students WHERE student_grade[1] > 70;
```

	student_id [PK] integer	student_name character varying (100)	student_grade integer[]
1	1	s1	{90,2020}
2	2	s2	{80,2019}

17.17 Array vs JSONB

Array Advantages:

1. Easy to set up
2. Requires less storage compared to JSON
3. Has multi-dimensional support (requires same number of elements though)
4. Indexing with GIN which greatly speeds up query performance
5. The PostgreSQL planner is likely to make better decisions with the PostgreSQL array, as it collects statistics on its contents but does not with JSONB

Array Disadvantages:

1. Limited to one datatype
2. Have to follow strict order of the array data input

Use Cases:

1. Simple lists of information. Especially useful when duplications are needed, for example for saving multiple ratings for the same value.

JSONB = JSON Binary

JSONB Advantages:

1. Very similar to the json data type
2. It provides additional operators for querying
3. Support for indexing for query performance
4. Syntactically, the JSONB array may be easier to use as you don't have to wrap your query value in a dummy array constructor.

`WHERE jsonb_column ? 'abc'` vs `WHERE textarraycolumn @> ARRAY['abc']`

Array Disadvantages:

1. PostgreSQL has to parse the JSONB data into a binary format
2. It tends to be slower than JSON data type when writing but faster when reading the data
3. Note that JSONB does not necessarily keep the order in the objects the same so it might not work all of the time because the order of items might change on the output.

`key2=>value2,key1=>value1`

Use Cases:

1. Any case of storing JSON type of data

17.18 Single & Multi-Dimensional Arrays

There's a number of data types available in PostgreSQL. There are times when you might want to **store multiple values** in one database column instead of across multiple tables. PostgreSQL gives you the capability with the **array data type**.

- Arrays are some of the most useful data types for **storing lists of information**. Whether you have a list of baseball scores, blog tags, or favourite book titles, arrays can be found everywhere. The bigger question, however, is why you'd even consider using an arrays. The answer comes down to performance and ease of use.
- Aggregating and joining data across tables and rows can be difficult, and depending on the type of data your storing, it might get expensive too.
- PostgreSQL gives the opportunity to **define a column of a table as a variable length single or multidimensional array**.
- Arrays of any built-in or **user defined base type, enum type or composite type** can be created.
- Arrays can be used to denormalise data and avoid lookup tables. A good rule of thumb for using them that way is that you mostly use the array as a whole, even if you might at times search for elements in the array.
- Heavier processing is going to be more complex than a lookup table.
- Think of an array as An array is essentially a group or list of items'.

- PostgreSQL allows a table column to **contain multi-dimensional arrays** that can be of any built-in or user defined data type.
- Arrays have an advantage over large plain text fields in that **data remains in a discreet and addressable form**. PostgreSQL also has a lot of functions dealing with arrays.

One dimensional Array

- It is a list of some values bundled together.
- It allows random access and all the elements can be accessed with the help of their index.
- The size of the array is fixed.

Multi-dimensional Array

- It also allows random access and all the elements can be accessed with the help of their index.
- It can also be seen as a collection of 1D arrays. It is also known as the **Matrix**.
- It's dimension can be increased from 2,3,4 and so on
- Store a list of lists of the element
- Represent multiple data items as a table consisting of rows and columns
- Multi-dimensional arrays are used to implement matrices.

18. JSON with PostgreSQL

[18.1 What is JSON?](#)

[18.2 JSON Syntax](#)

[18.3 JSON & JSONB data types](#)

[18.4 Exploring JSON Objects](#)

[18.5 Create table with JSONB data type](#)

[18.6 Update & delete JSON data](#)

[18.7 Create JSON from Tables](#)

[18.8 Use JSON AGG to Aggregate Data](#)

[18.9 Build a JSON array](#)

[18.10 Creating a Document from Data](#)

[18.11 Null Values in JSON Documents](#)

[18.12 JSON Nulls & SQL Nulls](#)

[18.13 Get Information from JSON Document](#)

[18.14 The Existence Operator](#)

[18.15 The Containment Operator](#)

[18.16 JSON Search with PostgreSQL functions](#)

[18.17 Indexing on JSONB](#)

18.1 What is JSON?

1. JSON stands for “JavaScript Object Notification” which is a type of data format popularly used by web applications.
2. JSON was introduced as an alternative to the XML format. In the “good old days” the data used to get transmitted in XML format which is a heavy weight data type compared to JSON.
3. JSON is lightweight format for storing and transporting data
4. JSON is a syntax for storing and exchanging data
5. JSON is “self-describing” and easy to understand, it is human readable.
6. The name behind popularising the JSON is Douglas Crockford. He used JSON in his company state software around 2001. In 2005, Yahoo started using JSON in its web services. In later 2006, Google started offering JSON in its gdata web protocol.
7. JSON is often used to serialise and transfer data over a network connection, for example between the web server and a web application. In computer science serialisation is a process of transforming data structures and objects in a format suitable to be stored in a file or memory buffer or transmitted over a network connection. Later on, this data can be retrieved. Because of the very nature of JSON, it is useful for storing or representing semi structured data.
8. JSON has got wide acceptance from the developer community. Today, JSON is one of the most widely used data-interchange formats in the web, and supported by most of the web’s API’s. to fetch public data and create applications out of them.

18.2 JSON Syntax

1. Data is in name/value pairs where a name value pair consists of a field name in double quotes followed by a colon followed by a value. Name:value.
2. Data is separated by commas.
3. Curly braces hold objects.
4. Objects can contain multiple name:value pairs.
5. Square brackets hold JSON arrays
6. Strings, numbers, Booleans and null can be used as values
 - a. String – “Title”:”Lord of the Rings”
 - b. Numbers – “price”:20.59
 - c. Booleans – “best seller”: true
 - d. Null – “discount”: null
7. Both Objects and arrays can be nested.

```
{  
  "firstname":"Elliott",  
  "country":"spain",  
  "employeeID":"1"  
}
```

```
"contacts": [  
  {  
    "firstname":"Elliott",  
    "country":"spain",  
    "employeeID":"1"  
  },  
  {  
    "firstname":"John",  
    "country":"USA",  
    "employeeID":"100"  
  }  
]
```

```
{  
  "title":"lord of the rings",  
  "author":"J.R.R.",  
  "details": {  
    "publisher":"ABC  
publishing"  
  },  
  "price": [  
    {  
      "type":"paperback"  
      "price":20.59  
    },  
    {  
      "type":"kindle"  
      "price":9.99  
    }  
  ]  
}
```

18.3 JSON & JSONB data types

1. PostgreSQL has become more-and-more popular since the JSON data type was introduced.
2. The application can store JSON strings in the PostgreSQL database using standard JSON format. Developers just need to tell the application to send across the JSON strings to the database as JSON data type and retrieve back in JSON format
3. PostgreSQL supports native JSON data type since version 9.2. It provides many functions and operators for manipulating JSON data.

JSON	JSONB
Pretty much like a TEXT data type which stores only valid JSON	stores the JSON in Binary format (JSONB)
Does not support FULL-TEXT-SEARCH Indexing	Supports FULL-TEXT-SEARCH indexing
Does not support wide range of JSON functions and operators	Supports all JSON functions and operators
JSON is faster as it does not process the input data	JSONB is faster on operations and functions, as it is pre-process and parse input data

18.4 Exploring JSON Objects

1. How is a JSON object represented in PostGreSQL?

```
SELECT '{"title":"lord of the Rings"}';
```

?column?	
text	
1	{\"title\":\"lord of the Rings\"}

A json object is represented as a curly braced named pair within single quotes.

Note how it is showing as a text data type.

2. Do I have to cast data types to make it a JSON?

```
SELECT '{
  "title":"lord of the Rings"
}':JSON;
```

json	
json	
1	{

1	{
2	"title":"lord of the Rings"
3	}

To specifically get it as JSON it has to be typecast to JSON. If I click on the cell body, a pop up will show the full contents of JSON in that cell.

3. Can we preserve formatting white spaces, if yes how?

```
SELECT '{
  "title":"lord of the Rings"
}':JSONB;
```

jsonb	
jsonb	
1	{\"title\": \"lord of the Rings\"}

Note how JSONB removes the formatting whitespaces whereas JSON preserves the formatting white spaces

18.5 Create table with JSONB data type

```
CREATE TABLE books (  
    book_id SERIAL PRIMARY KEY,  
    book_info JSONB  
);
```

```
INSERT INTO books (book_info) VALUES  
(  
    {  
        "title": "book title1",  
        "Author": "author1"  
    }  
);
```

```
INSERT INTO books (book_info) VALUES  
(  
    {  
        "title": "book title2",  
        "Author": "author2"  
    }  
,  
(  
    {  
        "title": "book title3",  
        "Author": "author3"  
    }  
,  
(  
    {  
        "title": "book title4",  
        "Author": "author4"  
    }  
);
```

book_info

General Definition Constraints Variables Security SQL

Data type jsonb

Length/Preci

```
SELECT * FROM books;
```

	book_id [PK] integer	book_info jsonb
1	1	{ "title": "book title1", "Author": "author1" }

```
SELECT * FROM books;
```

	book_id [PK] integer	book_info jsonb
1	1	{ "title": "book title1", "Author": "author1" }
2	2	{ "title": "book title2", "Author": "author2" }
3	3	{ "title": "book title3", "Author": "author3" }
4	4	{ "title": "book title4", "Author": "author4" }

	?column? jsonb
1	"book title1"
2	"book title2"
3	"book title3"
4	"book title4"

Use arrow notation to get the values from one named value pair. Note that I specify the name in single quotes. The return is in double quotes because when we select using single quotes the return is always in double quotes

```
SELECT book_info->'title' FROM books;
```

	?column? text
1	book title1
2	book title2
3	book title3
4	book title4

To get the return without double quotes, use a double arrow (->>)

```
SELECT book_info->>'title' FROM books;
```

	Title text	author text
1	book title1	author1
2	book title2	author2
3	book title3	author3
4	book title4	author4

```
SELECT
book_info->>'title' AS "Title",
book_info->>'Author' AS "author"
FROM books;
```

```
SELECT
book_info->>'title' AS "Title",
book_info->>'author' AS "author"
FROM books;
```

	Title text	author text
1	book title1	[null]
2	book title2	[null]
3	book title3	[null]
4	book title4	[null]

When specifying the name of the pair in the JSONB, it is case sensitive. Eg. 'Author' does return data but 'author' does not.

```
SELECT
book_info->>'title' AS "Title",
book_info->>'Author' AS "author"
FROM books
WHERE author = "author2";
ERROR: column "author" does not exist
LINE 5: WHERE author = "author2";
                        ^
SQL state: 42703 Character: 91
```

```
SELECT
book_info->>'title' AS "Title",
book_info->>'Author' AS "author"
FROM books
WHERE book_info->>'Author' = 'author2';
```

Use JSON format
in filter clauses

	Title text	author text
1	book title2	author2

18.6 Update & delete JSON data

```
INSERT INTO books (book_info) VALUES  
(  
  {  
    "title": "book title 10",  
    "Author": "author10"  
  }  
);
```

```
SELECT * FROM books;
```

	book_id [PK] integer	book_info jsonb
1	1	{"title": "book title1", "Author": "author1"}
2	2	{"title": "book title2", "Author": "author2"}
3	3	{"title": "book title3", "Author": "author3"}
4	4	{"title": "book title4", "Author": "author4"}
5	5	{"title": "book title 10", "Author": "author10"}

```
UPDATE books  
SET book_info = book_info || '{"Author": "Elliott"}'  
WHERE book_info->'Author' = 'author10';  
ERROR: Token "author10" is invalid.invalid input syntax for type json  
ERROR: invalid input syntax for type json  
SQL state: 22P02  
Detail: Token "author10" is invalid.  
Character: 94 Context: JSON data, line 1: author10
```

Use concatenation
|| operator which
will: add field or
replace existing
field

```
UPDATE books  
SET book_info = book_info || '{"Author": "Elliott"}'  
WHERE book_info->>'Author' = 'author10';  
UPDATE 1  
Query returned successfully in 59 msec.
```

The reason for the failure above is that the single arrow operator (->) returns data with double quotes. If I adjust the where clause to use the double arrow operator (->>) then it works

	book_id [PK] integer	book_info jsonb
1	1	{"title": "book title1", "Author": "author..."}
2	2	{"title": "book title2", "Author": "author..."}
3	3	{"title": "book title3", "Author": "author..."}
4	4	{"title": "book title4", "Author": "author..."}
5	5	{"title": "book title 10", "Author": "Elliott"}

```
SELECT * FROM books;
```

```
UPDATE books
SET book_info = book_info || '{"title":"something snappy"}'
WHERE book_info->>'Author' = 'Elliott'
RETURNING *;
```

Returning * will return all the affected rows.

	book_id [PK] integer	book_info jsonb
1	5	{"title": "something snappy", "Author": "Elliott"}

I can also update a JSONB field to contain an additional named value pair with a Boolean type.

```
UPDATE books
SET book_info = book_info || '{"Best Seller":true}'
WHERE book_info->>'Author' = 'Elliott'
RETURNING *;
```

	book_id [PK] integer	book_info jsonb
1	5	{"title": "something snappy", "Author": "Elliott", "Best Seller": true}

```
UPDATE books
SET book_info = book_info || '
{
  "Category": "science",
  "Additional Data": "bla blah",
  "pages": 250
}'
WHERE book_info->>'Author' = 'Elliott'
RETURNING *;
```

Multiple name:value pairs can be updated to a JSONB object.

	book_id [PK] integer	book_info jsonb
1	5	{"pages": 250, "title": "something snappy", "Author": "Elliott", "Category": "science", "Best Seller": true, "Additional Data": "bla blah"}

```
UPDATE books
SET book_info = book_info - 'Best Seller'
WHERE book_info->>'Author' = 'Elliott'
RETURNING *;
```

Use the minus – operator to remove a named pair from the JSONB object

	book_id [PK] integer	book_info jsonb
1	5	{ "pages": 250, "title": "something snappy", "Author": "Elliott", "Category": "science", "Additional Data": "bla blah" }

```
UPDATE books
SET book_info = book_info ||
'{"Availability Locations" : [
    "New York",
    "new Jersey"
]}'
WHERE book_info->>'Author' = 'Elliott'
RETURNING *;
```

Use the minus – operator to remove a named pair from the JSONB object

```
UPDATE books
SET book_info = book_info ||
'{"Availability Locations" : [
    "New York",
    "new Jersey"
]}'
WHERE book_info->>'Author' = 'Elliott'
RETURNING *;
```

I can also add a nested array within the object.

The screenshot shows a code editor with a toolbar at the top containing icons for menu, zoom, list, filter, search, undo, redo, and a 'Code' dropdown. The editor displays a JSONB object with the following structure:

```
1 {
2   "pages": 250,
3   "title": "something snappy",
4   "Author": "Elliott",
5   "Category": "science",
6   "Additional Data": "bla blah",
7   "Availability Locations": [
8     "New York",
9     "new Jersey"
10  ]
11 }
```

The status bar at the bottom indicates 'Ln: 1 Col: 1'. At the bottom right, there are 'Cancel' and 'OK' buttons.


```
UPDATE books
SET book_info = book_info #- '{Availability Locations,1}'
WHERE book_info->>'Author' = 'Elliott'
RETURNING *;
```

	book_id [PK] integer	book_info jsonb
1	5	{"pages": 250, "title": "something snappy", "Author": "Elliott", "Category": "science", "Additional Data": "bla blah", "Availability Locations": ["New York"]}

The name:value for “Availability Location” is an array type [] that contains two values. JSON arrays are indexed from zero. i.e. the first element of the array is zero, the second element is 1, the third element is 2 etc.

I can target an element within the array using its position and the #- to remove it from the array. Note above I am removing the second element of the array “availability Location” which has it's position denoted by 1.

18.7 Create JSON from Tables

Lets output directors table in JSON format.

```
SELECT * FROM directors;
```

	director_id [PK] integer	first_name character varying (150)	last_name character varying (150)	date_of_birth date	nationality character varying (20)	add_date date	update_date date
1	1	Tomas	Alfredson	1965-04-01	Swedish	[null]	[null]
2	2	Paul	Anderson	1970-06-26	American	[null]	[null]
3	3	Wes	Anderson	1969-05-01	American	[null]	[null]

ROW_TO_JSON function can convert a table or column(S) to JSON

```
SELECT ROW_TO_JSON(directors) FROM directors;
```

	row_to_json json
1	{"director_id":1,"first_name":"Tomas","last_name":"Alfredson","date_of_birth":"1965-04-01","nationality":"Swedish","add_date":null,"update_date":null}
2	{"director_id":2,"first_name":"Paul","last_name":"Anderson","date_of_birth":"1970-06-26","nationality":"American","add_date":null,"update_date":null}
3	{"director_id":3,"first_name":"Wes","last_name":"Anderson","date_of_birth":"1969-05-01","nationality":"American","add_date":null,"update_date":null}

Use a sub query to select specific row to return as an alias (t) for the row_to_json query.

```
SELECT ROW_TO_JSON(t) FROM  
(  
  SELECT  
    director_id,  
    first_name,  
    last_name,  
    nationality  
  FROM directors  
) AS t
```

	row_to_json json
1	{"director_id":1,"first_name":"Tomas","last_name":"Alfredson","nationality":"Swedish"}
2	{"director_id":2,"first_name":"Paul","last_name":"Anderson","nationality":"American"}
3	{"director_id":3,"first_name":"Wes","last_name":"Anderson","nationality":"American"}
4	{"director_id":4,"first_name":"Richard","last_name":"Ayoade","nationality":"British"}
5	{"director_id":5,"first_name":"Luc","last_name":"Besson","nationality":"French"}
6	{"director_id":6,"first_name":"James","last_name":"Cameron","nationality":"American"}

18.8 Use JSON_AGG to Aggregate Data

```
SELECT
director_id,
first_name || ' ' || last_name AS "name",
Date_of_birth,
nationality,
(
    SELECT JSON_AGG(x) AS all_movies FROM
    (
        SELECT movie_name
        FROM movies
        WHERE director_id = directors.director_id
    ) AS x
)
FROM directors;
```

List all movies for each director

Using JSON_AGG to aggregate all the movies for each director in the sub query.

	director_id [PK] integer	name text	date_of_birth date	nationality character varying (20)	all_movies json
1	1	Tomas Alfredson	1965-04-01	Swedish	[{"movie_name": "Let the Right One In"}]
2	2	Paul Anderson	1970-06-26	American	[{"movie_name": "There Will Be Blood"}]
3	3	Wes Anderson	1969-05-01	American	[{"movie_name": "Grand Budapest Hotel"},
4	4	Richard Ayoade	1977-06-12	British	[{"movie_name": "Submarine"}]
5	5	Luc Besson	1959-03-18	French	[{"movie_name": "Leon"},
6	6	James Cameron	1954-08-16	American	[{"movie_name": "Titanic"}]
7	7	Guillermo del Toro	1964-10-09	Mexican	[{"movie_name": "Pans Labyrinth"}]
8	8	Victor Fleming	1889-02-23	American	[{"movie_name": "Gone with the Wind"},
9	9	Francis Ford Coppola	1939-04-07	American	[{"movie_name": "Apocalypse Now"}]
10	10	Kinji Fukasaku	1930-07-03	Japanese	[{"movie_name": "Battle Royale"}]

18.9 Build a JSON array

Build a JSON array. Note that the `build_json_array` function takes any and mixed data types.

	json_build_array json
1	[1, 2, 3, 4, 5, 6, "Hi"]

```
SELECT JSON_BUILD_ARRAY(1,2,3,4,5);
```

	json_build_array json
1	[1, 2, 3, 4, 5]

```
SELECT JSON_BUILD_OBJECT(1,2,3,4,5,6,'Hi');  
ERROR: argument list must have even number of elements  
HINT: The arguments of json_build_object() must consist of alternating keys and values.  
SQL state: 22023
```

```
SELECT JSON_BUILD_OBJECT(1,2,3,4,5,6,'Hi',7);
```

`Build_json_object` function take an even number of items to make key:value pairs where the odd items are the key and the even items are the values.

	json_build_object json
1	{ "1" : 2, "3" : 4, "5" : 6, "Hi" : 7 }

I can manually supply key:pair data to the `json_build_object` function

```
SELECT JSON_BUILD_OBJECT(  
  'name','elliott',  
  'email','test@gmail.com'  
);
```

	json_build_object json
1	{ "name" : "elliott", "email" : "test@gmail.com" }

```
SELECT JSON_OBJECT('{name,email}','{elliott,test@gmail.com}');
```

`Json_objects( {keys}, {values} )`

	json_object json
1	{ "name" : "elliott", "email" : "test@gmail.com" }

`Json_object` function will also create an object where the keys are in curly braces followed by the values in curly braces.

18.10 Creating a Document from Data

```
CREATE TABLE directors_docs (  
  id SERIAL PRIMARY KEY,  
  body JSONB  
);
```



A screenshot of a SQL editor interface. The editor shows a JSON document being inserted into a table. The JSON is a single object with the following fields: "director_id" (3), "first_name" ("Wes"), "last_name" ("Anderson"), "date_of_birth" ("1969-05-01"), "nationality" ("American"), and "all_movies" (an array of three objects, each with a "movie_name" field). The movies listed are "Grand Budapest Hotel", "Rushmore", and "The Darjeeling Limited". The editor has a toolbar at the top with icons for undo, redo, and other editing functions. At the bottom, there is a status bar showing "Ln: 1 Col: 1" and a "Cancel" button.

```
1 {  
2   "director_id": 3,  
3   "first_name": "Wes",  
4   "last_name": "Anderson",  
5   "date_of_birth": "1969-05-01",  
6   "nationality": "American",  
7   "all_movies": [  
8     {  
9       "movie_name": "Grand Budapest Hotel"  
10    },  
11    {  
12      "movie_name": "Rushmore"  
13    },  
14    {  
15      "movie_name": "The Darjeeling Limited"  
16    }  
17  ]  
18 }
```

Ln: 1 Col: 1

Cancel

I can build a table of JSONB where each row contains details about the director then an array of all_movies by that director.

```
SELECT ROW_TO_JSON(a)::JSONB FROM  
(  
  SELECT  
    director_id,  
    first_name,  
    last_name,  
    date_of_birth,  
    nationality,  
    (  
      SELECT JSON_AGG(x) AS all_movies FROM  
      (  
        SELECT movie_name  
        FROM movies  
        WHERE director_id = directors.director_id  
      ) AS x  
    )  
  FROM directors  
) as a;
```

	row_to_json jsonb
1	{"last_name": "Alfredson", "all_movies": [{"movie_name": "Let the Right One In"}], "first_name": "Tomas", "director_id": 1, "nationality": "Swedish", "date_of_birth": "1965-04-01"}
2	{"last_name": "Anderson", "all_movies": [{"movie_name": "There Will Be Blood"}], "first_name": "Paul", "director_id": 2, "nationality": "American", "date_of_birth": "1970-06-26"}
3	{"last_name": "Anderson", "all_movies": [{"movie_name": "Grand Budapest Hotel"}, {"movie_name": "Rushmore"}, {"movie_name": "The Darjeeling Limited"}], "first_name": "Wes", "director_id": 3, "nationality": "American", "date_of_birth": "1969-05-01"}

```

INSERT INTO directors_docs (body)
SELECT ROW_TO_JSON(a)::JSONB FROM
(
  SELECT
    director_id,
    first_name,
    last_name,
    date_of_birth,
    nationality,
    (
      SELECT JSON_AGG(x) AS all_movies FROM
      (
        SELECT movie_name
          FROM movies
         WHERE director_id = directors.director_id
      ) AS x
    )
  FROM directors
) as a;
INSERT 0 38
Query returned successfully in 56 msec.

```

The advantage of this is that the database can return JSON directly to the request service rather than returning rows of SQL data which then has to be parsed by a script. This is particularly useful for feeding API responses.

	body jsonb
1	{"last_name": "Alfredson", "all_movies": [{"movie_name": "Let the Right One In"}], "first_name": "Tomas", "director_id": 1, "nationality": "Swedish", "date_of_birth": "1965-04-01"}
2	{"last_name": "Anderson", "all_movies": [{"movie_name": "There Will Be Blood"}], "first_name": "Paul", "director_id": 2, "nationality": "American", "date_of_birth": "1970-06-26"}
3	{"last_name": "Anderson", "all_movies": [{"movie_name": "Grand Budapest Hotel"}, {"movie_name": "Rushmore"}, {"movie_name": "The Darjeeling Limited"}], "first_name": "Wes", "di
4	{"last_name": "Ayuda", "all_movies": [{"movie_name": "Submarine"}], "first_name": "Richard", "director_id": 4, "nationality": "British", "date_of_birth": "1977-06-10"}

```
SELECT * FROM directors_docs;
```

I wrap the query into an insert query to add the JSON objects for each row into the directors_docs table body column.

```

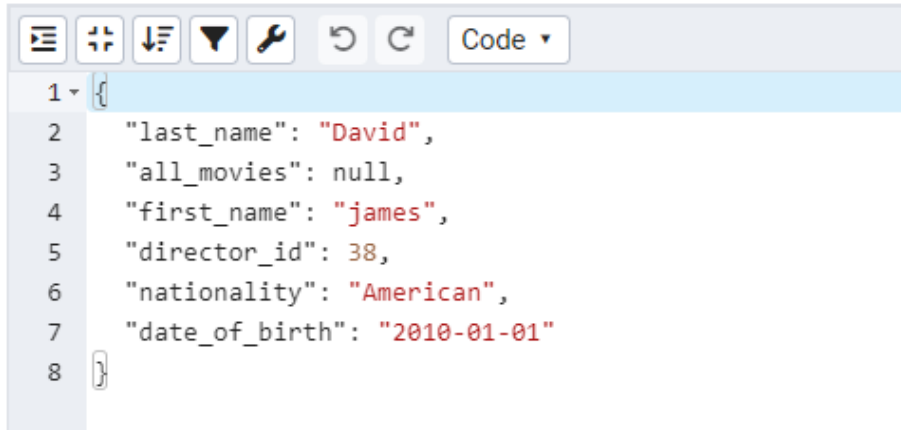
1 {
2   "last_name": "Anderson",
3   "all_movies": [
4     {
5       "movie_name": "Grand Budapest Hotel"
6     },
7     {
8       "movie_name": "Rushmore"
9     },
10    {
11      "movie_name": "The Darjeeling Limited"
12    }
13  ],
14  "first_name": "Wes",
15  "director_id": 3,
16  "nationality": "American",
17  "date_of_birth": "1969-05-01"
18 }

```

Ln: 1 Col: 1

Cancel OK

18.11 Null Values in JSON Documents



```
1 {
2   "last_name": "David",
3   "all_movies": null,
4   "first_name": "James",
5   "director_id": 38,
6   "nationality": "American",
7   "date_of_birth": "2010-01-01"
8 }
```

```
SELECT
JSONB_ARRAY_ELEMENTS(body->'all_movies')
FROM directors_docs;
ERROR: cannot extract elements from a scalar
SQL state: 22023
```

```
SELECT
JSONB_ARRAY_ELEMENTS(body->'all_movies')
FROM directors_docs
WHERE (body->'all_movies') IS NOT NULL;
ERROR: cannot extract elements from a scalar
SQL state: 22023
```

From the director_docs table the last director has no movies, i.e. **all_movies : null**.

If I try and use SQL to extract the all_movies element from the body JSONB field, I get an error.

```
DELETE FROM directors_docs;
```

Now I delete all rows on the directors_docs table.

Rebuild the query to count on each movie for each director.

```
SELECT * FROM directors_docs;
```

	id [PK] integer	body jsonb
1	39	{"count": 1, "last_name": "Alfredson", "first_
2	40	{"count": 1, "last_name": "Anderson", "first_
3	41	{"count": 3, "last_name": "Anderson", "first_
4	42	{"count": 1, "last_name": "Avonade", "first_n

```
INSERT INTO directors_docs (body)
SELECT ROW_TO_JSON(a)::JSONB FROM
(
  SELECT
  director_id,
  first_name,
  last_name,
  date_of_birth,
  nationality,
  (
    SELECT COUNT(x) FROM
    (
      SELECT movie_name
      FROM movies
      WHERE director_id =
directors.director_id
    ) AS x
  ) AS count
  FROM directors
) as a;
```

```

INSERT INTO directors_docs (body)
SELECT ROW_TO_JSON(a)::JSONB FROM
(
  SELECT
    director_id,
    first_name,
    last_name,
    date_of_birth,
    nationality,
    (
      SELECT
        CASE COUNT(x)
        WHEN 0 THEN '[]'
        ELSE JSON_AGG(x)
        END AS all_movies
      FROM
        (
          SELECT movie_name
          FROM movies
          WHERE director_id = directors.director_id
        ) AS x
    )
  FROM directors
) as a;

```

```

1 {
2   "last_name": "Anderson",
3   "all_movies": [
4     {
5       "movie_name": "Grand Budapest Hotel"
6     },
7     {
8       "movie_name": "Rushmore"
9     },
10    {
11      "movie_name": "The Darjeeling Limited"
12    }
13  ],
14  "first_name": "Wes",
15  "director_id": 3,
16  "nationality": "American",
17  "date_of_birth": "1969-05-01"
18 }

```

```

1 {
2   "last_name": "David",
3   "all_movies": [],
4   "first_name": "James",
5   "director_id": 38,
6   "nationality": "American",
7   "date_of_birth": "2010-01-01"
8 }

```

Now I have modified the query to include a case statement that returns an empty array [] if the count is zero or the JSON_AGG. Note the two JSON samples: one with an empty array, the other with an array of movies.

	body jsonb
1	{"last_name": "Alfredson", "all_movies": [{"movie_name": "Let the Right One In"}], "first_name": "Tomas", "director_id": 1, "nationality": "Swedish", "date_of_birth": "1965-04-01"}
2	{"last_name": "Anderson", "all_movies": [{"movie_name": "There Will Be Blood"}], "first_name": "Paul", "director_id": 2, "nationality": "American", "date_of_birth": "1970-06-26"}
3	{"last_name": "Anderson", "all_movies": [{"movie_name": "Grand Budapest Hotel"}, {"movie_name": "Rushmore"}, {"movie_name": "The Darjeeling Limited"}], "first_name": "Wes", "dir
4	{"last_name": "Ayuda", "all_movies": [{"movie_name": "Submarine"}], "first_name": "Richard", "director_id": 4, "nationality": "British", "date_of_birth": "1977-06-10"}

Now if I run the same query again it works. Conclusion is that it is very important to handle NULLS when creating JSON data.

```
SELECT
JSONB_ARRAY_ELEMENTS(body->'all_movies')
FROM directors_docs;
```

	jsonb_array_elements jsonb	
1	{"movie_name": "Let the Right One In"}	
2	{"movie_name": "There Will Be Blood"}	
3	{"movie_name": "Grand Budapest Hotel"}	
4	{"movie_name": "Rushmore"}	
5	{"movie_name": "The Darjeeling Limited"}	

18.12 JSON Nulls & SQL Nulls

JSON nulls are not exactly the same as SQL nulls.

- In SQL, a zero length string ("") is distinct from an SQL null value, which represents the **absence of a definite value**.
- In JSON, null is an actual value and is **represented by a JSON literal ("null")**.

JSON nulls must be distinguishable from SQL nulls. SQL/JSON syntax enables the application author to select whether they should be omitted from the JSON object or array being constructed.

18.13 Get Information from JSON Document

```
SELECT
*,
JSONB_ARRAY_LENGTH(body->'all_movies') AS total_movies
FROM directors_docs
ORDER BY JSONB_ARRAY_LENGTH(body->'all_movies') DESC;
```

I can get all columns plus a count of all_movies by each row (director) and order this data. Note the function JSONB_ARRAY_LENGTH which is performing the count.

	id [PK] integer	body jsonb	total_movies integer
1	165	{"last_name": "Kubrick"...	3
2	176	{"last_name": "Raimi", "...	3
3	155	{"last_name": "Anderso...	3
4	169	{"last_name": "Lucas", "...	3
5	178	{"last_name": "Scorses...	3

```
SELECT
*,
JSONB_OBJECT_KEYS(body)
FROM directors_docs;
```

This query will list all the keys for each JSON object.

	id [PK] integer	body jsonb	jsonb_object_keys text
1	153	{"last_name": "Alfredson", "all_movies": [{"movie_name": "Let the Ri...	last_name
2	153	{"last_name": "Alfredson", "all_movies": [{"movie_name": "Let the Ri...	all_movies
3	153	{"last_name": "Alfredson", "all_movies": [{"movie_name": "Let the Ri...	first_name
4	153	{"last_name": "Alfredson", "all_movies": [{"movie_name": "Let the Ri...	director_id
5	153	{"last_name": "Alfredson", "all_movies": [{"movie_name": "Let the Ri...	nationality
6	153	{"last_name": "Alfredson", "all_movies": [{"movie_name": "Let the Ri...	date_of_birth
7	154	{"last_name": "Anderson", "all_movies": [{"movie_name": "There Wil...	last_name
8	154	{"last_name": "Anderson", "all_movies": [{"movie_name": "There Wil...	all_movies
9	154	{"last_name": "Anderson", "all_movies": [{"movie_name": "There Wil...	first_name

```
SELECT
j.key,
j.value
FROM directors_docs, JSONB_EACH(directors_docs.body) AS j;
```

JSONB_EACH will get the key and the value of each key:value pair within the JSON object of each row of the table. I can then alias this as j and output key and value into a row of the query result.

I can output JSONB as a traditional SQL table (JSONB_TO_RECORD) but I need to define the columns.

	key text	value jsonb
1	last_name	"Alfredson"
2	all_movies	[{"movie_name": "Let the Right One In"}]
3	first_name	"Tomas"
4	director_id	1
5	nationality	"Swedish"
6	date_of_birth	"1965-04-01"
7	last_name	"Anderson"
8	all_movies	[{"movie_name": "There Will Be Blood"}]
9	first_name	"Paul"
10	director_id	2
11	nationality	"American"

```
SELECT
j.*
FROM directors_docs, JSONB_TO_RECORD(directors_docs.body) AS j;
ERROR: a column definition list is required for functions returning "record"
LINE 3: FROM directors_docs, JSONB_TO_RECORD(directors_docs.body) AS...
                                ^
SQL state: 42601 Character: 33
```

```
SELECT
j.*
FROM directors_docs,
JSONB_TO_RECORD(directors_docs.body) AS j
(
    director_id INT,
    first_name VARCHAR(255),
    last_name VARCHAR(255),
    date_of_birth DATE,
    nationality VARCHAR(255),
);
```

	director_id integer	first_name character varying (255)	last_name character varying (255)	date_of_birth date	nationality character varying (255)
1	1	Tomas	Alfredson	1965-04-01	Swedish
2	2	Paul	Anderson	1970-06-26	American
3	3	Wes	Anderson	1969-05-01	American
4	4	Richard	Ayoade	1977-06-12	British
5	5	Luc	Besson	1959-03-18	French
6	6	James	Cameron	1954-08-16	American
7	7	Guillermo	del Toro	1964-10-09	Mexican
8	8	Victor	Fleming	1889-02-23	American
9	9	Francis	Ford Coppola	1939-04-07	American
10	10	Kinji	Fukasaku	1920-07-03	Japanese

18.14 The Existence Operator

```
SELECT
*
FROM directors_docs
WHERE body->'first_name' ? 'John';
```

Find all first_name
equal to "John"

	id [PK] integer	body jsonb
1	166	{"last_name": "Lasseter", "all_movies": [{"movie_name": "Toy Story"}], "first_name": "John", "d

```
1 {
2   "last_name": "Lasseter",
3   "all_movies": [
4     {
5       "movie_name": "Toy Story"
6     }
7   ],
8   "first_name": "John",
9   "director_id": 14,
10  "nationality": "American",
11  "date_of_birth": "1957-01-12"
12 }
```

We can search in all node names to find any JSON object where the value of a key:value pair matches a specific value. Note that when using the Existence operator (?) I must use a specific node or it will not work.

```
SELECT
*
FROM directors_docs
WHERE body->'director_id' = ? '1';
```

	id [PK] integer	body jsonb

The existence operator expects both sides to be a TEXT value. In this case the left side is text but the right side is numeric so it does not work. – use the Containment operator for numeric values.

18.15 The Containment Operator

```
SELECT
*
FROM directors_docs
WHERE body @> '{"first_name":"John"}';
```

	id [PK] integer	body jsonb
1	166	{"last_name": "Lasseter", "all_movies": [{"movie_name": "Toy Story"}], "first_name": "John", "d

The containment operator (@>) can be used to match on specific key:value's within the JSON objects.

```
1 {
2   "last_name": "Lasseter",
3   "all_movies": [
4     {
5       "movie_name": "Toy Story"
6     }
7   ],
8   "first_name": "John",
9   "director_id": 14,
10  "nationality": "American",
11  "date_of_birth": "1957-01-12"
12 }
```

```
SELECT
*
FROM directors_docs
WHERE body @> '{"director_id":1}';
```

	id [PK] integer	body jsonb
1	153	{"last_name": "Alfredson", "all_movies": [{"movie_name": "Let the Right One In"}], "first_name": "Tomas", "director_id": 1, "na

Note that director_id is a numeric data type so I have excluded the double quotes. If I run it with "1" then it will treat director_id values as strings and return nothing.

```

SELECT
*
FROM directors_docs
WHERE body->'all_movies' @> '{"movie_name":"Toy Story"}';
ERROR: Expected string or "}", but found "[".invalid input syntax for type json
ERROR: invalid input syntax for type json
SQL state: 22P02
Detail: Expected string or "}", but found "[".
Character: 61
Context: JSON data, line 1: {[...

```

```

SELECT
*
FROM directors_docs
WHERE body->'all_movies' @> '[{"movie_name":"Toy Story"}]';

```

	id [PK] integer	body jsonb
1	166	{"last_name": "Lasseter", "all_movies": [{"movie_name": "Toy Story"}], "fir

Find the record for movie name “toy Story”

Movies are inside an array called ‘all_movies’ so first on the left hand side I have defined that.

Now I have fixed the right hand side to put the square brackets outside of the curly braces that specify the key:value pair.

18.16 JSON Search with PostgreSQL functions

```
SELECT
*
FROM directors_docs
WHERE body->'first_name' ? 'J%';
```

id	body
[PK] integer	jsonb

Find all records where first name starts with “J”

The coalescing operator will return nothing.

```
SELECT
*
FROM directors_docs
WHERE body->'first_name' LIKE 'J%';
ERROR: operator does not exist: jsonb ~~ unknown
LINE 4: WHERE body->'first_name' LIKE 'J%';
                        ^
HINT: No operator matches the given name and argument
types. You might need to add explicit type casts.
SQL state: 42883
Character: 58
```

I have to use like but this also fails because I have to use the double arrow selector to return the data without the double quotes otherwise it will look for exact matches of first_name equal to 'J%'.

```
SELECT
*
FROM directors_docs
WHERE body->>'first_name' LIKE 'J%';
```

	id	body
	[PK] integer	jsonb
1	158	{"last_name": "Cameron", "all_movies": [{"movie_name": "Titanic"}], "first_name": "James", "dire
2	166	{"last_name": "Lasseter", "all_movies": [{"movie_name": "Toy Story"}], "first_name": "John", "dir
3	171	{"last_name": "McTeigue", "all_movies": [{"movie_name": "V for Vendetta"}], "first_name": "Jam

```
SELECT
*
FROM directors_docs
WHERE (body->>'director_id')::INTEGER > 2;
```

Find all records where director_id is greater than 2

	id	body
	[PK] integer	jsonb
1	155	{"last_name": "Anderson", "all_movies": [{"movie_name": "Grand Budapest Hotel"}, {"movie_name": "Rushmore"}], {
2	156	{"last_name": "Ayoade", "all_movies": [{"movie_name": "Submarine"}], "first_name": "Richard", "director_id": 4, "nati
3	157	{"last name": "Besson", "all movies": [{"movie name": "Leon"}, {"movie name": "The Fifth Element"}], "first name"

```
SELECT
*
FROM directors_docs
WHERE (body->>'director_id')::INTEGER IN(1,2,3,45,10);
```

Find all records where
director_id is in array of values

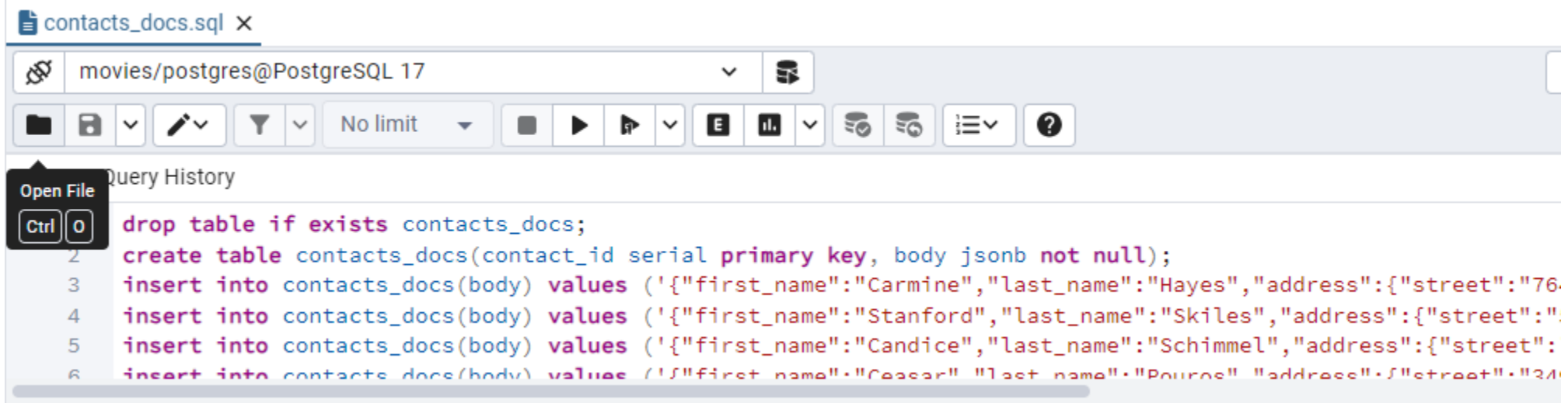
	id [PK] integer	body jsonb
1	153	{"last_name": "Alfredson", "all_movies": [{"movie_name": "Let the Right One In"}], "first_name": "Tomas", "director_id": 1, "nat
2	154	{"last_name": "Anderson", "all_movies": [{"movie_name": "There Will Be Blood"}], "first_name": "Paul", "director_id": 2, "nat
3	155	{"last_name": "Anderson", "all_movies": [{"movie_name": "Grand Budapest Hotel"}, {"movie_name": "Rushmore"}, {"movie
4	162	{"last_name": "Fukasaku", "all_movies": [{"movie_name": "Battle Royale "}], "first_name": "Kinji", "director_id": 10, "nationa

```
EXPLAIN SELECT
*
FROM directors_docs
WHERE body->>'first_name' LIKE 'J%';
```

	QUERY PLAN
	text
1	Seq Scan on directors_docs (cost=0.00..3.57 rows=1 width=226)
2	Filter: ((body ->> 'first_name'::text) ~~ 'J%'::text)

Using the explain keyword will output a text table of what the query is doing

18.17 Indexing on JSONB



The screenshot shows a PostgreSQL client interface with a tab labeled 'contacts_docs.sql'. The Query History pane displays the following SQL commands:

```
drop table if exists contacts_docs;  
create table contacts_docs(contact_id serial primary key, body jsonb not null);  
insert into contacts_docs(body) values ('{"first_name": "Carmine", "last_name": "Hayes", "address": {"street": "7641 Stephan Estates", "city": "Considineview", "state": "CA", "zip": "94025"}}');  
insert into contacts_docs(body) values ('{"first_name": "Stanford", "last_name": "Skiles", "address": {"street": "503 Dulce Rue", "city": "West Samton", "state": "CA", "zip": "94025"}}');  
insert into contacts_docs(body) values ('{"first_name": "Candice", "last_name": "Schimmel", "address": {"street": "7641 Stephan Estates", "city": "Considineview", "state": "CA", "zip": "94025"}}');  
insert into contacts_docs(body) values ('{"first_name": "Caesar", "last_name": "Pouros", "address": {"street": "7641 Stephan Estates", "city": "Considineview", "state": "CA", "zip": "94025"}}');
```

Use the open icon to open the 20,000 row SQL query of contacts_docs and run the query.

```
NOTICE: table "contacts_docs" does not exist, skipping  
INSERT 0 1  
Query returned successfully in 922 msec.
```

From the output I can see that it is showing rows 1 to 1000 out of 20 pages of rows. I have an ID column and a body column containing JSONB.

```
SELECT * FROM contacts_docs;
```

Data Output			Messages	Notifications
	contact_id [PK] integer	body jsonb		
1	1	{ "address": { "city": "Considineview", "street": "7641 Stephan Estates", "last_name": "Hayes", "first_name": "Carmine" }		
2	2	{ "address": { "city": "West Samton", "street": "503 Dulce Rue", "last_name": "Skiles", "first_name": "Stanford" }		

```
EXPLAIN ANALYZE SELECT
*
FROM contacts_docs
WHERE body @> '{"first_name": "John"}';
```

I can use the keywords of EXPLAIN & ANALYZE to get a table of data about the query. Note That I am searching in 20,000 JSON objects to find matching records.

	QUERY PLAN
	text
1	Seq Scan on contacts_docs (cost=0.00..665.00 rows=2 width=135) (actual time=0.708..4.548 rows=9 loops=1)
2	Filter: (body @> '{"first_name": "John"}'::jsonb)
3	Rows Removed by Filter: 19991
4	Planning Time: 0.114 ms
5	Execution Time: 4.563 ms

Note the **planning time** (how long it takes PostgreSQL to plan the execution) then the **execution time** (how long it took to run the query).

Note that I can run this query multiple times and the times will vary slightly. It is also telling me that it is performing a sequential scan. There are no indexes on this data.

Can the query execution time be sped up? Yes with a GIN index.

GIN stands for Generalised Inverted Index.

The core capability of GIN Index is to speed up full text searches.

When performing a search based on specific keys or elements in a text or document, GIN index is the way to go.

GIN index stores “key” (or an element or value) and the “position list” pairs. The position list is the row ID of the key. This means, if the key occurs at multiple places in the document, GIN index stores the key only once along with it’s position of occurrences which not only keeps the GIN index compact in size but also helps speed up searching.

This is an enhancement from PostgreSQL version 9.4 onwards.

```
CREATE INDEX idx_gin_contacts_docs_body ON contacts_docs USING GIN(body);
CREATE INDEX
Query returned successfully in 254 msec.
```

```
CREATE INDEX index_name ON table_name USING GIN(column_name)
```

In PgAdmin 4 → Database → table I can see the index.

```
EXPLAIN ANALYZE SELECT
*
FROM contacts_docs
WHERE body @> '{"first_name": "John"}';
```

Without indexing the query took 4.563ms to run.
With GIN indexing the query has completed in half the time at **0.276ms.**

Tables (10)
actors
contacts_docs
Columns
Constraints
Indexes (1)
idx_gin_contacts_docs_body
RLS Policies
Rules
Triggers

QUERY PLAN	text
1	Bitmap Heap Scan on contacts_docs (cost=21.55..29.16 rows=2 width=135) (actual time=0.231..0.249 rows=9 loops=1)
2	Recheck Cond: (body @> '{"first_name": "John"}':jsonb)
3	Heap Blocks: exact=9
4	-> Bitmap Index Scan on idx_gin_contacts_docs_body (cost=0.00..21.55 rows=2 width=0) (actual time=0.207..0.207 rows=9 loops=1)
5	Index Cond: (body @> '{"first_name": "John"}':jsonb)
6	Planning Time: 1.282 ms
7	Execution Time: 0.276 ms

```
SELECT pg_size_pretty(pg_relation_size('idx_gin_contacts_docs_body'::regclass)) AS index_name;
```

index_name	text
1	3664 kB

The index is 3664kb (almost 3Mb) of data has been added as index data. Indexing can be expensive in terms of increasing the database size.

```
CREATE INDEX idx_gin_contacts_docs_body_cool ON contacts_docs USING GIN(body JSONB_PATH_OPS);
```

There is a better index for JSON data using GIN JSONB_PATH_OPS

```
SELECT pg_size_pretty(pg_relation_size('idx_gin_contacts_docs_body_cool'::regclass)) AS  
index_name;
```

	index_name	
	text	🔒
1	2512 kB	

Note how this index is 2512Kb (2.5Mb) compared to almost 3Mb previously so this has shaved about half a Mb of the index size.

Can we create an index on a specific JSON key? - Yes

```
CREATE INDEX idx_gin_contacts_docs_body_fname ON contacts_docs USING GIN((body->'first_name')  
JSONB_PATH_OPS);
```

```
SELECT pg_size_pretty(pg_relation_size('idx_gin_contacts_docs_body_fname'::regclass)) AS  
index_name;
```

	index_name	
	text	🔒
1	288 kB	

This index has come out significantly smaller because I am only indexing one key:pair of the JSON objects.

19. Indexes & performance Optimisation

[19.1 What is an Index?](#)

[19.2 Create an Index](#)

[19.3 Create Unique Indexes](#)

[19.4 List all Indexes](#)

[19.5 Size of the Table Index](#)

[19.6 List Count of all Indexes](#)

[19.7 Drop an Index](#)

[19.8 SQL Statement Execution Process](#)

[19.9 SQL Statement Execution Stages](#)

[19.10 The Query Optimizer](#)

[19.11 Optimizer Node Types](#)

[19.12 Sequential Nodes](#)

[19.13 Index Nodes](#)

[19.14 Join Nodes](#)

[19.15 Index Types – B-Tree Index](#)

[19.16 Index Types – Hash Index](#)

[19.17 Index Types – Brin Index](#)

[19.18 Index Types – GIN Index](#)

19. Indexes & performance Optimisation

[19.19 The Explain Statement](#)

[19.20 Explain Output Options](#)

[19.21 Using Explain Analyze](#)

[19.22 Understand Query Cost Model](#)

[19.23 Indexes Are Not Free](#)

[19.24 Indexes for Sorted Output](#)

[19.25 Using multiple Indexes on a single query](#)

[19.26 Execution Plans Depend on Input Values](#)

[19.27 Using Organised vs Random Data](#)

[19.28 Try to use Index Only Scan](#)

[19.29 Partial Indexes](#)

[19.30 Expression Index](#)

[19.31 Adding Data while Indexing](#)

[19.32 Invalidating an Index](#)

[19.33 Rebuilding an Index](#)

19.1 What is an Index?

An index is a structured relation.

An index helps improve the access to data in the database.

Indexed tuples point to the table page where the row is stored.

An index is a **data structure** that allows **faster access to the underlying table** so that specific tuples can be found quickly. Here “quickly” means faster than scanning the whole underlying table and analysing every single tuple.

Maintaining an index is a **fundamental** for having good performance.

Performance tuning is one of the **most complex tasks** in the job of a database administrator.

Adding indexes **may improve the speed of the data access** but they add a **COST** to the data modification. Therefore it is important to understand if the index is used.

Indexes are placed on table column or columns.

Why not put an index on all columns? Too many indexes will result in slow insert/update/delete operations. (high cost).

PostgreSQL supports indexes of up to 32 columns.

Two main types of Index:

- INDEX – create an index on only values of a column or Columns
- UNIQUE INDEX – Create an index on only UNIQUE values of a column or columns.

19.2 Create an Index

```
CREATE INDEX index_name ON table_name (col1, col2, ...)
```

```
CREATE UNIQUE INDEX index_name ON table_name (col1, col2, ...)
```

```
CREATE INDEX index_name ON table_name (using method)
```

```
Column_name [ASC|DESC] [NULLS (FIRST|LAST)],
```

Index syntax

Try to keep the naming conventions unique and globally accessible. i.e.

```
INDEX          CREATE INDEX idx_table_name_column_name_col2
```

```
UNIQUE INDEX   CREATE INDEX idx_u_table_name_column_name
```



```
CREATE INDEX idx_orders_order_date ON orders (order_date);  
CREATE INDEX  
Query returned successfully in 66 msec.
```

In PgAdmin 4 → Database → Table → Indexes

Idx_orders_order_date

General Definition Columns SQL

Access Method: btree

With

Fill factor

Gin pending list limit

Pages per range

Buffering

Deduplicate items?

Fast update?

Autosummarize?

Unique?

NULLs not distinct?

Clustered?

Constraint: 1

Access method is btree which is the default

Idx_orders_order_date

General Definition Columns SQL

Columns/Expressions

Col/Exp	Operator class	Sort order	NULLs	Collation	Statistics
C order_date		ASC	LAST		

Include columns

Close Reset Save

The definition of the Index can be seen where things like unique are not selected.

In the columns tab I can see default params of sort order and Nulls.

```
CREATE INDEX idx_orders_ship_city ON orders (ship_city);  
CREATE INDEX  
Query returned successfully in 47 msec.
```

Single column index

```
CREATE INDEX idx_orders_customer_id_order_id ON orders (customer_id, order_id);  
CREATE INDEX  
Query returned successfully in 61 msec.
```

Multi column
index

The screenshot shows the 'Columns' tab of a PostgreSQL index configuration window. The index name is 'idx_orders_customer_id_order_id'. The 'Columns/Expressions' table lists two columns: 'customer_id' and 'order_id', both with 'ASC' sort order and 'LAST' NULLs placement. The 'Collation' for 'customer_id' is 'pg_catalog."default"'. Below the table is an 'Include columns' section with an empty text input field.

	Col/Exp	Operator class	Sort order	NULLs	Collation	Statistics
C	customer_id		ASC	LAST	pg_catalog."default"	
C	order_id		ASC	LAST		

Include columns

It is important to note that, when creating multi-column indexes, you should always place the most selective column first. PostgreSQL will consider a multicolumn index from the first column onward, so if the first column is the most selective, the index access method will be the cheapest.

- Order of the field and how you are using it in the where clause

To create an index in PgAdmin 4:

Right click on indexes → Create Index

General tab → Fill out Index name and set tablespace to 'pg-default'

Definition tab → Default is btree, other methods available such as heap, hash, GIN, etc

Columns tab → select column(s) → Add then adjust sort order and Nulls

Click save button.

 idx_shippers_company_name ×

General Definition Columns SQL

Columns/Expressions

	Col/Exp	Operator class	Sort order	NULLs	Collation	Statistics
C	company_name		ASC	LAST	pg_catalog."default"	

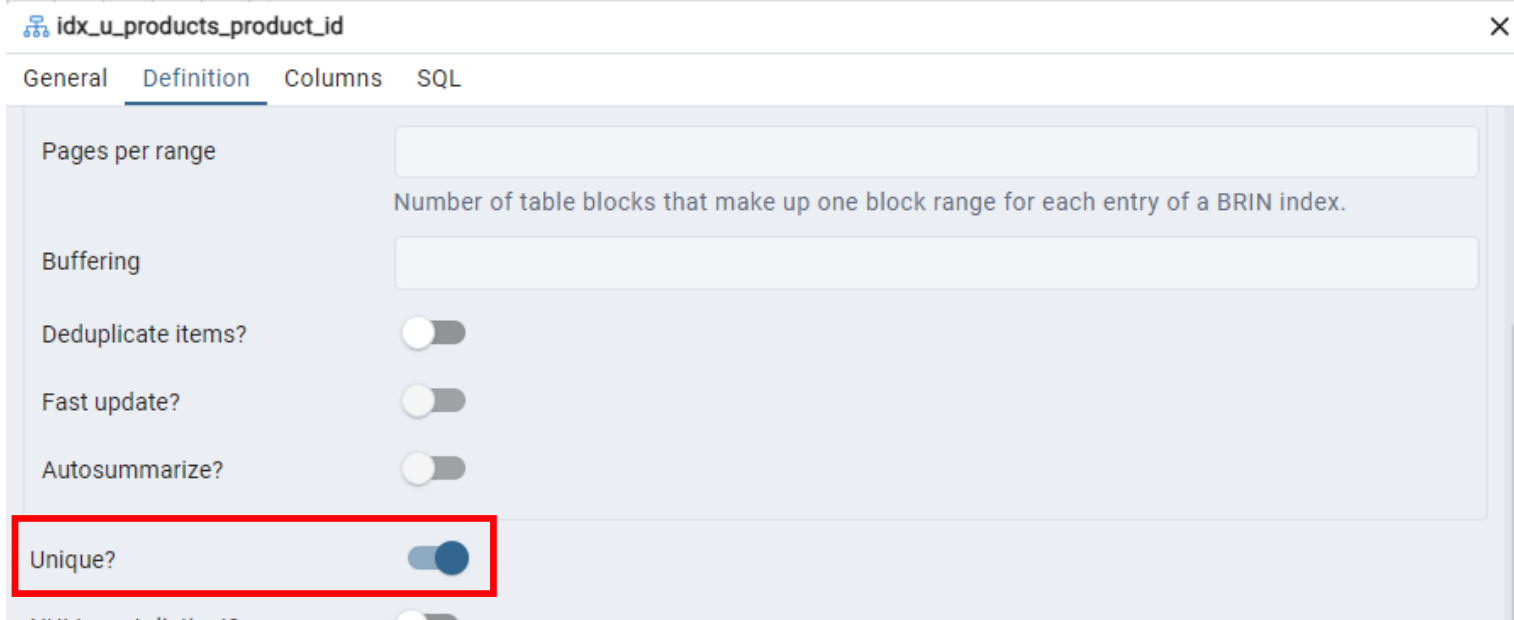
Include columns

19.3 Create Unique Indexes

Normally the primary key of a table are kept with a UNIQUE INDEX.

If you combine a UNIQUE index for two or more columns, the combined values in these columns cannot be duplicated in multiple rows.

```
CREATE UNIQUE INDEX idx_u_products_product_id ON products (product_id);  
CREATE INDEX  
Query returned successfully in 71 msec.
```



```
CREATE UNIQUE INDEX idx_u_employees_employee_id ON employees (employee_id);  
CREATE INDEX  
Query returned successfully in 53 msec.
```

```
CREATE UNIQUE INDEX idx_u_orders_order_id_customer_id ON orders (order_id, customer_id);  
CREATE INDEX  
Query returned successfully in 71 msec.
```

```
CREATE UNIQUE INDEX idx_u_employees_employee_id_hire_date ON employees (employee_id, hire_date);  
CREATE INDEX  
Query returned successfully in 36 msec.
```

`SELECT * FROM employees;` The unique key is on 'employee_id' & 'hire_date'

Data Output Messages Notifications									
Showing rows: 1 to 9 Page No: 1 of 1									
	employee_id [PK] smallint	last_name character varying (20)	first_name character varying (10)	title character varying (30)	title_of_courtesy character varying (25)	birth_date date	hire_date date	address character varying (60)	
1	1	Davolio	Nancy	Sales Representative	Ms.	1948-12-08	1992-05-01	507 - 20th Ave. E.\nApt. 2	
2	2	Fuller	Andrew	Vice President, Sales	Dr.	1952-02-19	1992-08-14	908 W. Capital Way	
3	3	Leverling	Janet	Sales Representative	Ms.	1963-08-30	1992-04-01	722 Moss Bay Blvd.	
4	4	Peacock	Margaret	Sales Representative	Mrs.	1937-09-19	1993-05-03	4110 Old Redmond Rd.	

```
INSERT INTO employees (employee_id,first_name,last_name, hire_date)  
VALUES (1, 'A', 'B', '1992-05-01');  
ERROR: Key (employee_id)=(1) already exists.duplicate key value violates unique  
constraint "pk_employees"  
ERROR: duplicate key value violates unique constraint "pk_employees"  
SQL state: 23505  
Detail: Key (employee_id)=(1) already exists.
```

This insert fails not because of the index defined above but because the employee_id is the primary key and must be unique.

```
CREATE TABLE t1 (  
    id SERIAL PRIMARY KEY,  
    tag TEXT  
);
```

```
INSERT INTO t1 (tag) values ('a'),('b');
```

```
CREATE UNIQUE INDEX idx_u_t1_id_text ON t1 (tag);
```

```
INSERT INTO t1 (tag) values ('a');  
ERROR: Key (tag)=(a) already exists.duplicate key value violates unique constraint  
"idx_u_t1_id_text"  
ERROR: duplicate key value violates unique constraint "idx_u_t1_id_text"  
SQL state: 23505  
Detail: Key (tag)=(a) already exists.
```

If I have a unique index on a column I cannot have duplicates in that column.

19.4 List all Indexes

`SELECT * FROM pg_indexes;` Pg_indexes table contains all the indexes of a database

`SELECT * FROM pg_indexes WHERE schemaname = 'public';` list indexes in the public schema only

	schemaname name	tablename name	indexname name	tablespace name	indexdef text
1	public	categories	pk_categories	[null]	CREATE UNIQUE INDEX pk_categories ON public.categories USING btree (category_id)
2	public	customer_customer_demo	pk_customer_customer_demo	[null]	CREATE UNIQUE INDEX pk_customer_customer_demo ON public.customer_customer_demo USING btree
3	public	customer_demographics	pk_customer_demographics	[null]	CREATE UNIQUE INDEX pk_customer_demographics ON public.customer_demographics USING btree (cu
4	public	customers	pk_customers	[null]	CREATE UNIQUE INDEX pk_customers ON public.customers USING btree (customer_id)
5	public	employees	pk_employees	[null]	CREATE UNIQUE INDEX pk_employees ON public.employees USING btree (employee_id)

`SELECT * FROM pg_indexes WHERE schemaname = 'public' ORDER BY tablename, indexname;`

I can order the public indexes by tablename then index name.

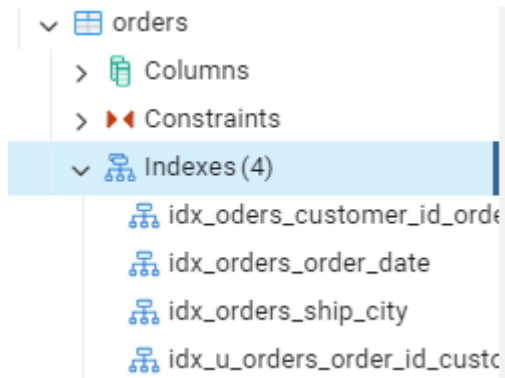
`SELECT * FROM pg_indexes WHERE tablename = 'orders' ORDER BY indexname;` filter indexes by tablename.

Data Output

Messages

Notifications

19.5 Size of the Table Index



orders
Columns
Constraints
Indexes (4)
idx_orders_customer_id_order_date
idx_orders_order_date
idx_orders_ship_city
idx_u_orders_order_id_customer_id

```
SELECT pg_indexes_size('orders');
```

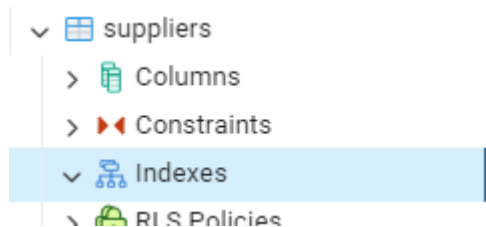
The 'orders' table has 4 indexes created and I can view the total size of all table indexes. The value is shown in bytes but I can view this in a prettier format.

```
SELECT pg_size_pretty(pg_indexes_size('orders'));
```

	pg_indexes_size bigint
1	172032

	pg_size_pretty text
1	168 kB

The orders table only has 830 rows of data and has produced an index of 168kb. Imagine that a table has 10 million rows, the sizes of the indexes can become very large which has a cost so **always ensure that the indexes are being used.**



suppliers
Columns
Constraints
Indexes
RLS Policies

```
SELECT pg_size_pretty(pg_indexes_size('suppliers'));
```

	pg_size_pretty text
1	16 kB

The 'suppliers' table does not have any indexes so the total index size is low at 16kb. The reason for this small footprint is that there is a PK constraint defined on this table.

```
CREATE INDEX idx_suppliers_region ON suppliers (region);  
CREATE INDEX  
Query returned successfully in 69 msec.
```

Index size increase
from 16kb to 32kb

```
SELECT pg_size_pretty(pg_indexes_size('suppliers'));
```

	pg_size_pretty text
1	32 kB


```
CREATE UNIQUE INDEX idx_suppliers_supplier_id ON suppliers (supplier_id);
CREATE INDEX
Query returned successfully in 68 msec.
```

```
SELECT pg_size_pretty(pg_indexes_size('suppliers'));
```

	pg_size_pretty text
1	48 kB

Now with two indexes, the 'suppliers' table with only 29 rows has a total index size of 48Kb. Indexes will grow over time as more and more data is added to a database which occupies disk space on the server.

19.6 List Count of all Indexes

Adding indexes may improve the speed of the data access but they add a COST to the data modification. Therefore it is important to understand of they are being used.

```
SELECT * FROM pg_stat_all_indexes;
```

	relid oid	indexrelid oid	schemaname name	relname name	indexrelname name	idx_scan bigint	last_idx_scan timestamp with time zone	idx_tup_read bigint	idx_tup_fetch bigint
1	16669	16670	pg_toast	pg_toast_16666	pg_toast_16666_index	0	[null]	0	0
2	16674	16675	pg_toast	pg_toast_16671	pg_toast_16671_index	0	[null]	0	0
3	16679	16680	pg_toast	pg_toast_16676	pg_toast_16676_index	0	[null]	0	0
4	16684	16685	pg_toast	pg_toast_16681	pg_toast_16681_index	0	[null]	0	0

This table is a list of statistics for each index showing a count for how many times they have been read, fetched, scanned etc.

I can filter the list of indexes to the specific schema and order the rows

```
SELECT * FROM pg_stat_all_indexes WHERE schemaname = 'public' ORDER BY relname, indexrelname;
```

	relid oid	indexrelid oid	schemaname name	relname name	indexrelname name	idx_scan bigint	last_idx_scan timestamp with time zone	idx_tup_read bigint	idx_tup_fetch bigint
1	16666	16726	public	categories	pk_categories	0	[null]	0	0
2	16671	16728	public	customer_customer_demo	pk_customer_customer_demo	0	[null]	0	0
3	16676	16730	public	customer_demographics	pk_customer_demographics	0	[null]	0	0
4	16681	16732	public	customers	pk_customers	0	[null]	0	0
5	16691	16736	public	employee_territories	pk_employee_territories	0	[null]	0	0
6	16696	74581	public	employees	idx_u_employees_employee_id	0	[null]	0	0

Use 'relname' to filter index stats by table.

```
SELECT * FROM pg_stat_all_indexes WHERE relname = 'orders' ORDER BY indexrelname;
```

Showing rows: 1 to 5 Page No: 1 of 1									
	relid oid	indexrelid oid	schemaname name	relname name	indexrelname name	idx_scan bigint	last_idx_scan timestamp with time zone	idx_tup_read bigint	idx_tup_fetch bigint
1	16697	74577	public	orders	idx_oders_customer_id_order_id	0	[null]	0	0
2	16697	74575	public	orders	idx_orders_order_date	0	[null]	0	0
3	16697	74576	public	orders	idx_orders_ship_city	0	[null]	0	0
4	16697	74582	public	orders	idx_u_orders_order_id_customer_id	0	[null]	0	0
5	16697	16740	public	orders	pk_orders	0	[null]	0	0

These SQL commands can be used to see which indexes are being used to optimise the database and remove low or no usage indexes.

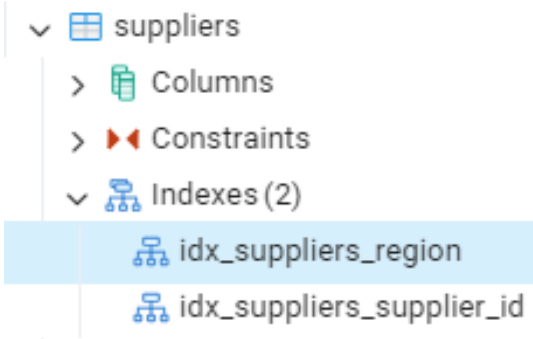
19.7 Drop an Index

```
DROP INDEX [ CONCURRENTLY]
[IF EXISTS] index_name
[CASCADE | RESTRICT];
```

CASCADE - If the index has dependants, you can use the CASCADE option to automatically drop these objects and all objects that depend on these objects.

RESTRICT – The Restrict option instructs PostgreSQL to refuse to drop an index if any objects depend on it. The drop index uses RESTRICT by default

CONCURRENTLY – when you execute a drop index statement, PostgreSQL acquires an exclusive lock on the table and block others from accessing it until the drop index removal completes.



```
DROP INDEX idx_suppliers_region;
DROP INDEX
Query returned successfully in 47 msec.
```

The 'idx_suppliers_region' index does not have any dependants so it can be dropped normally.

If dropping an index concurrently on a very large table then this will involve an outage because the table will be unavailable until the query statement completes so a planned works to inform all users may be needed.

19.8 SQL Statement Execution Process

SQL is a declarative language. This means that we ask the database to execute something on the data but we do not specify or tell the database how to do it. For example:

*SELECT * FROM orders WHERE order_id = 1;* I am not telling the database how to fetch the data from persistent storage, I am asking what data I would like with the clauses.

You trust your database to run the best way to fetch the data. PostgreSQL does this really well and handles workload well to get data fast. Fastest path vs time spent reasoning about this path. Sometimes PostgreSQL will not take the absolute fastest path but the path that is easiest to reason fast.

As a DBA, it's your job to provide as much information to the database engine to take the best route to get your result set.

19.9 SQL Statement Execution Stages

There are four main stages for SQL statement execution:

Parser – Handles textual form of the statement (the SQL text) and verifies whether it is correct or not. Disassemble into tables, columns, clauses etc.

Rewriter – Applying any syntactic rules to rewrite the original SQL statement.

Optimiser – Finding the fastest path to the data that the statement needs.

Executor – Responsible for going to the storage and retrieving/inserting/updating/deleting the data. Gets physical access to the data.

19.10 The Query Optimizer

The optimizer is the component that decides what to use to access the data and how quickly it can access the data.

The optimizer by default uses COST to decide how to access the data with the lowest cost chosen. After parsing and rewriting the optimizer looks at the cost of the relevant operation and **picks the lowest cost**. PostgreSQL has a cost to every operation. If an SQL statement asks to perform 4 operation PostgreSQL will add up all the costs for the operations to produce a total cost. The plan with the lowest cost is passed to the executor.

If the statement is complex and combines many tables where the time spent to calculate the cost would be overtaken by the time to execute the query then a generic operation is used to access the data. This compromise is between the time spent in computing the path to the data and finding a not so good path to access the data.

From PostgreSQL 9.6 onwards, the executor can also perform data access using parallel jobs. This means that retrieving a very large set of data can be divided between parallel workers known as **THREADS**. The optimizer divides a set of actions to pass to the executor into **NODES**. For example: `SELECT * FROM orders ORDER BY order_id;` - This SQL query has two operations;

1. All data – This might be handled by Node #1
2. Order by – This might be handled by Node #2

Node types: There are different types of node for each of the operations.

19.11 Optimizer Node Types

Nodes are available for;

- Every operation, and
- Every access method

Nodes are stackable;

Parent Node (cost)

Child Node 1 (cost)

Child Node 2 (cost)

...

The output of a node can be used as the input to another node. i.e the output of child node 2 becomes the input of child node 1.

The nodes can produce a very complex plan to access the data. Each node will have a cost to perform the operation.

Types of nodes:

- Sequential Scan (The most by default used node allows to scan the entire table)
- Index scan, Index only scan and Bitmap Index Scan
- Nested loop, Hash join and Merge Join (Hash used for equality and comparison)
- The Gather and Merge parallel Nodes

For each type of node the access method and operation is different.

The end goal is to have the **lowest cost** to get to our data the fastest possible way.

This SQL statement will list all the nodes.

```
SELECT * FROM pg_am;
```

	oid [PK] oid	amname name	amhandler regproc	amtype "char"
1	2	heap	heap_tableam_handler	t
2	403	btree	bthandler	i
3	405	hash	hashhandler	i
4	783	gist	gisthandler	i
5	2742	gin	ginhandler	i
6	4000	spgist	spghandler	i
7	3580	brin	brinhandler	i

19.12 Sequential Nodes

Sequential scan is the default when no other valuable alternative.

Reads from the beginning of the dataset, block by block.

Filtering clauses is not very limiting, so the end result will be to get almost the whole table contents.

Sequential read-all operation is faster.

```
SELECT * FROM orders;
```

A select all statement would be done by a sequential scan because we want all rows and all columns of data so it needs to read it block by block.

```
SELECT * FROM orders WHERE order_id IS NOT NULL;
```

Also done by a sequential scan because the filtering clause is not very limiting.

Sequential scans are very expensive. They have a high cost and in a production environment should be avoided.

```
EXPLAIN SELECT * FROM orders;
```

QUERY PLAN	
text	
1	Seq Scan on orders (cost=0.00..22.30 rows=830 width=90)

Note that the explain keyword tells us that the query plan is using sequential scan (seq scan)

```
EXPLAIN SELECT * FROM orders WHERE order_id IS NOT NULL;
```

QUERY PLAN	
text	
1	Seq Scan on orders (cost=0.00..22.30 rows=830 width=90)

19.13 Index Nodes

An index is used to access the dataset

Data file and index files are separated but they are nearby

Index nodes scan types

- Index Scan Index → seeking the tuples → then read again the data
- Index Only Scan requested index columns only → directly to get data from index file
- Bitmap Index Scan Builds a memory bitmap of where tuples that satisfy the statement clauses

An Index Scan is expensive because there is a cost to seek the tuples then a cost to read the data.

An index only scan is less expensive because the requested column data is in the index so it is the fastest route.

```
EXPLAIN SELECT * FROM orders WHERE order_id = 1;
```

	QUERY PLAN	
	text	
1	Index Scan using idx_u_orders_order_id_customer_id on orders (cost=0.28..8.29 rows=1 width=90)	
2	Index Cond: (order_id = 1)	

This query uses an index Scan because the order_id is indexed in 'idx_u_orders_order_id_customer_id'. This route has a high cost of two trips to access the data.


```
EXPLAIN SELECT order_id FROM orders WHERE order_id = 1;
```

	QUERY PLAN text	
1	Index Only Scan using idx_u_orders_order_id_customer_id on orders (cost=0.28..8.29 rows=1 width=2)	
2	Index Cond: (order_id = 1)	

This query uses an index Only Scan because the order_id column is the only data which I want and it is in the index so the query plan only has to reference the index to find the desired data.

```
EXPLAIN SELECT order_id, ship_via FROM orders WHERE order_id = 1;
```

	QUERY PLAN text	
1	Index Scan using idx_u_orders_order_id_customer_id on orders (cost=0.28..8.29 rows=1 width=4)	
2	Index Cond: (order_id = 1)	

This query uses an index Scan because the order_id only is included in an index but the ship_via column is not in any index. The cost is first to go to the index then to go to the data.

19.14 Join Nodes

Used when joining the tables

Joins are performed on two tables at a time; if more tables are joined together, the output of one join is treated as the input to a subsequent join.

When joining a large number of tables, the generic query optimizer settings may affect what combinations of joins are considered.

Types:

Hash Join

- Inner table Builds a hash table from the inner table, keyed by the join key
- Outer table Then scans the outer table checking if a corresponding value is present.

```
SHOW work_mem;
```

	work_mem
	text
1	4MB

The working memory is defaulted to 4Mb and is the amount of memory dedicated to internal sort operation and hash table before writing to temporary disk file.

If more than one operation is running in parallel then each operation will be allowed 4Mb. Sort operations are order by, distinct, and merge join.

Hash tables → Hash joins, hash based aggregation and hash based processing such as IN queries.

```
SELECT * FROM orders WHERE order_id IN('')
```

Merge join

Joins two children sorted by their shared join key. This only needs to scan each relation once, but both inputs need to be sorted by the join key first (or scanned in a way that produces already-sorted output like an index scan matching the required sort order).

Nested Loop

For each row in the outer table, iterate through all the rows in the inner table and see if they match the join condition. If the inner relation can be scanned with an index, that can improve the performance of a nested loop Join. This is generally an inefficient way to process joins but it is always available and sometimes may be the only option.

```
EXPLAIN SELECT * FROM orders NATURAL JOIN customers;
```

	QUERY PLAN text	
1	Hash Join (cost=4.05..28.63 rows=830 width=210)	
2	Hash Cond: (orders.customer_id = customers.customer_id)	
3	-> Seq Scan on orders (cost=0.00..22.30 rows=830 width=90)	
4	-> Hash (cost=2.91..2.91 rows=91 width=126)	
5	-> Seq Scan on customers (cost=0.00..2.91 rows=91 wid...	

This query is using hash joins where the output of one hash is feeding the hash above.

19.15 Index Types – B-Tree Index

Default Index type/access method when creating a new index.

Self-balancing tree index. It keeps it's depth constant, even with a very large underlying table.

SELECT, INSERT, DELETE and sequential access in logarithmic time basis.

Can be used for most operators and column types.

Supports UNIQUE condition and normally used to build the PK indexes.

Used when columns involve the operators of '>', '<', '>=', '<=', BETWEEN, IN, IS NULL, IS NOT NULL

Used for pattern matching even for like based queries

ONE DRAWBACK: It copies the whole column or column values into the tree structure that you chose to index.

19.16 Index Types – Hash Index

The hash Index only works with the equality operator (only simple equality comparison with =)

Not for range or disqualify operators

Larger than b-tree indexes

```
CREATE INDEX index_name ON table_name  
USING hash (column_name);
```

```
CREATE INDEX idx_orders_order_date ON orders USING hash (order_date);  
CREATE INDEX  
Query returned successfully in 47 msec.
```

```
EXPLAIN SELECT * FROM orders WHERE order_date = '2020-01-01';
```

	QUERY PLAN text
1	Index Scan using idx_orders_order_date on orders (cost=0.00..8.02 rows=1 width=90)
2	Index Cond: (order_date = '2020-01-01'::date)

The query plan has chosen to use the create hash index 'idx_orders_order_date' because we are using an equality operator.

The hash index takes up more space than a b-tree index and can only be used for equality checks.

19.17 Index Types – BRIN Index

Block Range Index

Data block → min to max value

Smaller index

Less costly to maintain than b-tree index

Can be used on large tables vs b-tree index

Uses linear sort order, e.g. customer → order_date

19.18 Index Types – GIN Index

Generalized Inverted Indexes

Point to multiple tuples

Used with array data types

Used in full text search

Useful when we have multiple values stored in a single column

19.19 The Explain Statement

Shows the query execution plan

Shows the LOWEST COST among evaluated plans

Will not execute the statement you enter, just shows how it will execute the query

Shows the execution nodes that the executor will use to provide you with the dataset

```
EXPLAIN SELECT * FROM suppliers WHERE supplier_id = 1;
```

	QUERY PLAN text	
1	Seq Scan on suppliers (cost=0.00..1.36 rows=1 width=740)	
2	Filter: (supplier_id = 1)	

Type of execution node – Seq scan for single node execution

On Table – suppliers

Cost – starts from zero and ends at 1.36

Fitters that are running on the node

Size of tuple

Rows found – How many tuples

The cost is the amount of effort to execute the node. It is expressed with two values. The first value is the startup cost and represents how much work PostGreSQL has to do before it can begin executing the node. The second value represents the cost to provide the last bit of the data i.e. to fully execute the node.

Rows found is the number of tuples the node is expected to provide in the final dataset and is a pure estimation so the value may be incorrect, nor can the number of tuples be zero.

Size of tuple is how many bits every tuple will occupy as an average. This is used to estimate network traffic that the query will use.

```
EXPLAIN SELECT company_name FROM suppliers ORDER BY company_name;
```

	QUERY PLAN text
1	Sort (cost=1.99..2.07 rows=29 width=98)
2	Sort Key: company_name
3	-> Seq Scan on suppliers (cost=0.00..1.29 rows=29 width=98)

This query is using two nodes to complete. The nodes are obvious because they have a cost value.

Read the node stack from bottom up. Note how first it runs a sequential scan which is fed into the sort key. The sort node has a startup cost which is greater because it is accumulating the cost from the sequential scan.

Nodes execute a query plan in a pipeline and tells me that the sort cannot start until the seq scan has been completed.

```
EXPLAIN SELECT supplier_id FROM suppliers WHERE supplier_id = 1 ORDER BY company_name ,city;
```

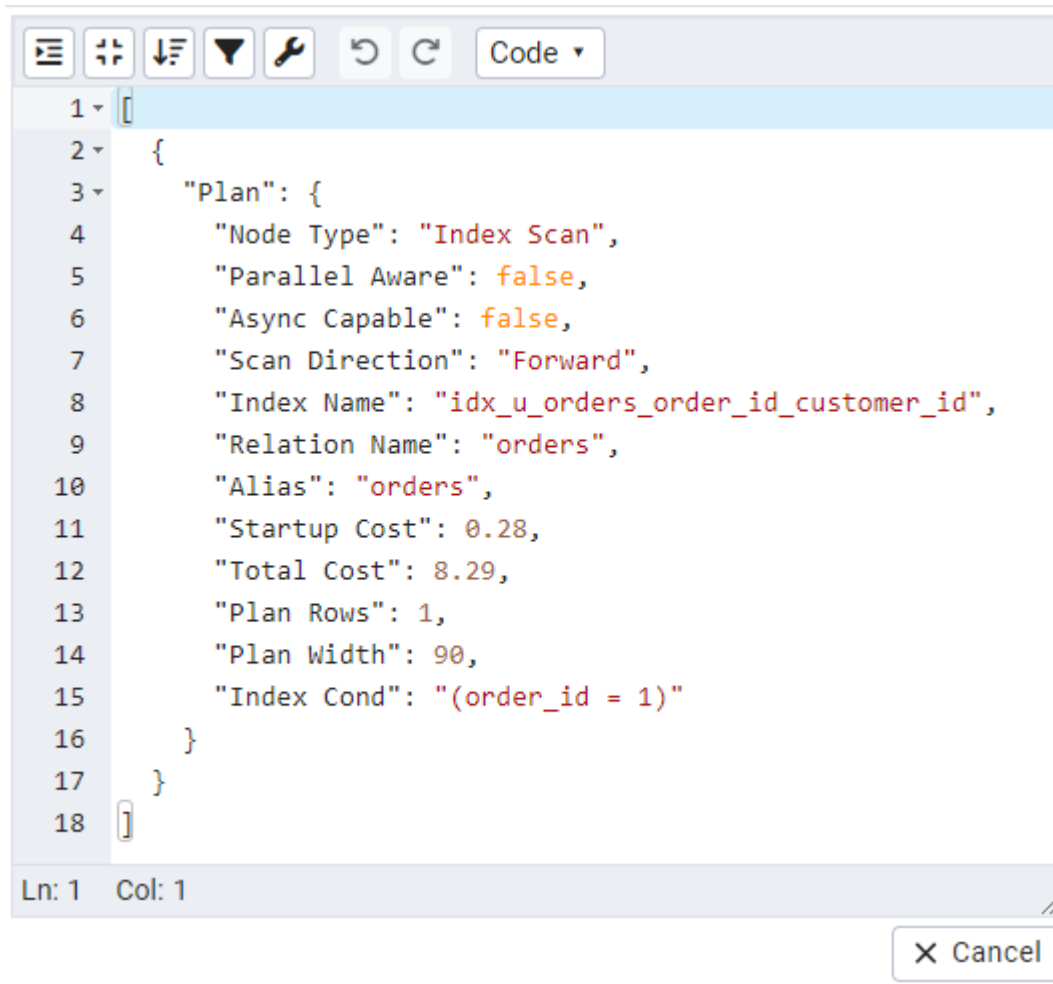
	QUERY PLAN text
1	Sort (cost=1.37..1.38 rows=1 width=148)
2	Sort Key: company_name, city
3	-> Seq Scan on suppliers (cost=0.00..1.36 rows=1 width=...
4	Filter: (supplier_id = 1)

For this query the filter is performed first. Then a seq scan is run on the suppliers table. Then this is passed to the sort node.

19.20 Explain Output Options

By default explain is output as text but it can also be output as XML, JSON or YAML

```
EXPLAIN ( FORMAT JSON) SELECT * FROM orders WHERE order_id = 1;
```



The screenshot shows a database client interface with a toolbar at the top containing icons for menu, expand, collapse, filter, settings, refresh, and a 'Code' dropdown. The main area displays the JSON output of the query, with line numbers 1 through 18 on the left. The JSON structure is as follows:

```
1 [
2   {
3     "Plan": {
4       "Node Type": "Index Scan",
5       "Parallel Aware": false,
6       "Async Capable": false,
7       "Scan Direction": "Forward",
8       "Index Name": "idx_u_orders_order_id_customer_id",
9       "Relation Name": "orders",
10      "Alias": "orders",
11      "Startup Cost": 0.28,
12      "Total Cost": 8.29,
13      "Plan Rows": 1,
14      "Plan Width": 90,
15      "Index Cond": "(order_id = 1)"
16    }
17  }
18 ]
```

At the bottom left, the status bar shows 'Ln: 1 Col: 1'. At the bottom right, there is a button labeled 'X Cancel'.

From a readability point of view the JSON output of the Query plan might be better for complex query analysis.

19.21 Using Explain Analyze

It prints out the best plan to execute the query

It runs the query

Also reports back some statistical information

```
EXPLAIN ANALYZE SELECT * FROM orders WHERE order_id = 1 ORDER BY order_id;
```

	QUERY PLAN
	text
1	Index Scan using idx_u_orders_order_id_customer_id on orders (cost=0.28..8.24 rows=1 width=90) (actual time=0.022..0.023 rows=0 loops=1)
2	Index Cond: (order_id = 1)
3	Planning Time: 0.126 ms
4	Execution Time: 0.041 ms

Type of node, i.e. indexing scan

Name of index used

Cost starts at 0.28 and ends at 8.24

Loops is number of times this node has been called

Actual startup time 0.022 to warm up, and execution time of 0.23. effectively this query took 1 ms to execute.

Planning time – how long the optimizer took to plan the execution of the query (ms)

Execution time – how long to execute the query (ms)

19.22 Understand Query Cost Model

```
CREATE TABLE t_big (  
  id SERIAL,  
  name TEXT  
);
```

This will create a table with
4 million rows of data

```
INSERT INTO t_big (name)  
SELECT 'adam' FROM generate_series(1,2000000);  
INSERT 0 2000000  
Query returned successfully in 4 secs 376 msec.
```

```
INSERT INTO t_big (name)  
SELECT 'linda' FROM generate_series(1,2000000);  
INSERT 0 2000000  
Query returned successfully in 4 secs 608 msec.
```

```
EXPLAIN SELECT * FROM t_big WHERE id = 12345;
```

	QUERY PLAN text	
1	Gather (cost=1000.00..43455.43 rows=1 width=9)	
2	Workers Planned: 2	
3	-> Parallel Seq Scan on t_big (cost=0.00..42455.33 rows=1 width=9)	
4	Filter: (id = 12345)	

```
SHOW max_parallel_workers_per_gather;
```

	max_parallel_workers_per_gather text	
1	2	

Note how the optimizer is using two workers in parallel to sequence scan all the rows the find the record.

Note the initial cost and final cost of parallel seq scan. This means two workers are combining to work on the filter together.

The partial results are then combined through the gather node which has a starting cost of 1000 when it receives from the parallel workers. The final cost is 43455.

```
SET max_parallel_workers_per_gather TO 0;  
Query returned successfully in 88 msec.
```

I have temporarily set parallel workers to zero to inhibit the use of parallel workers.

```
EXPLAIN SELECT * FROM t_big WHERE id = 12345;
```

	QUERY PLAN text
1	Seq Scan on t_big (cost=0.00..71622.00 rows=1 width=9)
2	Filter: (id = 12345)

The cost is an abstract concept to allow the query optimizer to evaluate different methods for completing the query.

```
SELECT pg_relation_size('t_big') / 8192.0;
```

The pg_relation_size function will return the size of the table in bytes / 8192 = block size where each block is 8192 bytes each.

	?column? numeric
1	21622.000000000000

	seq_page_cost text
1	1

```
SHOW seq_page_cost;
```

```
SHOW cpu_tuple_cost;
```

	cpu_tuple_cost text
1	0.01

```
SHOW cpu_operator_cost;
```

	cpu_operator_cost text
1	0.0025

```
SELECT 21622*1+ 4000000*0.01 + 4000000 * 0.0025;
```

	?column? numeric
1	71622.0000

The cost is calculated with the below formula using the above default values as well as the row count for the table.

$$\text{pg\_relation\_size} * \text{seq\_page\_cost} + \text{total\_number\_of\_table\_records} * \text{cpu\_tuple\_cost} + \text{total\_number\_of\_table\_records} * \text{cpu\_operator\_cost}$$

```
SET max_parallel_workers_per_gather TO 2;  
Query returned successfully in 59 msec.
```

19.23 Indexes are not Free

```
SELECT pg_size_pretty(pg_indexes_size('t_big'));
```

	pg_size_pretty text
1	0 bytes

Right now there are no indexes on the t_big table and the size of the indexes is zero.

```
SELECT pg_size_pretty(pg_total_relation_size('t_big'));
```

	pg_size_pretty text
1	169 MB

The table size with no indexes is 169mb for 4 million rows

```
EXPLAIN ANALYZE SELECT * FROM t_big WHERE id = 123456;
```

	QUERY PLAN text
1	Gather (cost=1000.00..43455.43 rows=1 width=9) (actual time=15.055..176.886 rows=1 loops=1)
2	Workers Planned: 2
3	Workers Launched: 2
4	-> Parallel Seq Scan on t_big (cost=0.00..42455.33 rows=1 width=9) (actual time=94.574..146.191 rows=0 loops=3)
5	Filter: (id = 123456)
6	Rows Removed by Filter: 1333333
7	Planning Time: 0.054 ms
8	Execution Time: 176.900 ms

The query is using two workers to run a sequential scan with a very high cost. The total time is planning time plus execution time. The cost is high because there is no index. In a production environment this would be a disaster because of the total run time.

```
CREATE INDEX idx_t_big_id ON t_big (id);
CREATE INDEX
Query returned successfully in 2 secs 229 msec.
```

The index took over 2 seconds to create because the table has 4 million rows.

```
SELECT pg_size_pretty(pg_indexes_size('t_big'));
```

	pg_size_pretty text
1	86 MB

The index size is 86Mb

	pg_size_pretty text
1	255 MB

```
SELECT pg_size_pretty(pg_total_relation_size('t_big'));
```

The table size with no indexes was 169Mb but with an index it has increased to 255Mb (168Mb plus 86Mb)

```
EXPLAIN ANALYZE SELECT * FROM t_big WHERE id = 123456;
```

	QUERY PLAN text
1	Index Scan using idx_t_big_id on t_big (cost=0.43..8.45 rows=1 width=9) (actual time=0.204..0.205 rows=1 loops=1)
2	Index Cond: (id = 123456)
3	Planning Time: 1.363 ms
4	Execution Time: 0.219 ms

Without index

planning time: 0.054ms

Execution time: 176.900ms

Total time: 176.954ms

With index

planning time: 1.364ms

Execution time: 0.219ms

Total time: 1.583ms

The cost of an index is that the writes to the table will be slower because the index needs to be sync'd with the table. However the gain is that the searching is considerably faster in large tables.

```
SHOW max_parallel_maintenance_workers;
```

Since PostgreSQL v11 more than one CPU in parallel can be used to build a b-tree index.

	max_parallel_maintenance_workers text
1	2

19.24 Indexes for Sorted Output

```
EXPLAIN SELECT * FROM t_big ORDER BY id LIMIT 20;
```

	QUERY PLAN text	🔒
1	Limit (cost=0.43..1.06 rows=20 width=9)	
2	-> Index Scan using idx_t_big_id on t_big (cost=0.43..125505.43 rows=4000000 width=9)	

There are two operations in this query (order by then limit). Note how it is using the index and the cost is still low. Note the warmup cost of 0.43 and the total cost of 1.06. This is because the index has returned the data in the right order so there is no sort node in the query plan making it faster to run.

```
EXPLAIN SELECT * FROM t_big ORDER BY name LIMIT 20;
```

	QUERY PLAN text	🔒
1	Limit (cost=83638.10..83640.43 rows=20 width=9)	
2	-> Gather Merge (cost=83638.10..472554.22 rows=3333334 width=9)	
3	Workers Planned: 2	
4	-> Sort (cost=82638.08..86804.74 rows=1666667 width=9)	
5	Sort Key: name	
6	-> Parallel Seq Scan on t_big (cost=0.00..38288.67 rows=1666667 width=9)	

If I do a sort on a column that is not indexed note how much more complex and time consuming the query planner is.

First it is running a parallel seq scan to sort using two workers. It is then passing this to the gather merge. The gather merge then passes the output to a limit.

Min and Max also work on sorted data.

```
EXPLAIN SELECT MIN(id), MAX(id) FROM t_big;
```

	QUERY PLAN text	
1	Result (cost=0.91..0.92 rows=1 width=8)	
2	InitPlan 1	
3	-> Limit (cost=0.43..0.46 rows=1 width=4)	
4	-> Index Only Scan using idx_t_big_id on t_big (cost=0.43..103884.43 rows=4000000 width=4)	
5	InitPlan 2	
6	-> Limit (cost=0.43..0.46 rows=1 width=4)	
7	-> Index Only Scan Backward using idx_t_big_id on t_big t_big_1 (cost=0.43..103884.43 rows=4000000 width=4)	

This query plan is using an index only scan backwards then a limit to 1 to find the max. It is then using an index scan (forward) and a limit to 1 to find the min.

Min and Max are perfect candidates for indexing so whenever using ORDER BY or MIN or MAX, on a column, these functions all use sorting and are ideal candidate for an index. Not how it is not referencing the table data but taking all the info from the index to return a result because the id column is indexed.

19.25 Using Multiple Indexes on a Single Query

```
EXPLAIN SELECT * FROM t_big WHERE id = 20 OR id = 40;
```

	QUERY PLAN	
	text	
1	Bitmap Heap Scan on t_big (cost=8.88..16.85 rows=2 width=9)	
2	Recheck Cond: ((id = 20) OR (id = 40))	
3	-> BitmapOr (cost=8.88..8.88 rows=2 width=0)	Main node
4	-> Bitmap Index Scan on idx_t_big_id (cost=0.00..4.44 rows=1 width=0)	
5	Index Cond: (id = 20)	Sub Node
6	-> Bitmap Index Scan on idx_t_big_id (cost=0.00..4.44 rows=1 width=0)	
7	Index Cond: (id = 40)	Sub Node

This query is to find two records so the query planner is using the same index twice. The main node is using two sub nodes and each sub node has a condition. The sub nodes are bitmap Index scans. Each sub node has an initial cost of 0 and a final cost of 4.44. In the main node the total cost is 0.88. the main node is combining the results of each sub node.

A bitmap scan is scanning blocks which contain pages of table. i.e.

Scan 1 → blocks – pages of table

Scan 2 → blocks – pages of table

...

The query has to look for two rows and the sub nodes may return multiple blocks so the **recheck condition is checking the bitmap return to ensure that the desired results are in the blocks.**

Once this is verified it is **passed to the bitmap heap scan.**

19.26 Execution Plans Depend on Input Values

```
CREATE INDEX idx_t_big_name ON t_big (name);  
CREATE INDEX  
Query returned successfully in 1 secs 765 msec.
```

```
EXPLAIN SELECT * FROM t_big WHERE name = 'adam' LIMIT 10;
```

	QUERY PLAN text
1	Limit (cost=0.00..0.36 rows=10 width=9)
2	-> Seq Scan on t_big (cost=0.00..71622.00 rows=1997333 width=9)
3	Filter: (name = 'adam'::text)

Note how in both these queries the query optimizer is choosing not to use the indexes but is instead using a sequence scan.

This is because half the names in the name column are 'adam' and the other half 'linda' so it is quicker to go straight to the table data.

```
EXPLAIN SELECT * FROM t_big WHERE name = 'adam' OR name = 'linda';
```

	QUERY PLAN text
1	Seq Scan on t_big (cost=0.00..81622.00 rows=3000002 width=9)
2	Filter: ((name = 'adam'::text) OR (name = 'linda'::text))

Rather than using the index. 'adam' and 'linda' make up the entire dataset and PostgreSQL know about this when it checked the statistics

```
EXPLAIN SELECT * FROM t_big WHERE name = 'adam1' OR name = 'linda1';
```

	QUERY PLAN text
1	Bitmap Heap Scan on t_big (cost=8.88..12.89 rows=1 width=9)
2	Recheck Cond: ((name = 'adam1'::text) OR (name = 'linda1'::text))
3	-> BitmapOr (cost=8.88..8.88 rows=1 width=0)
4	-> Bitmap Index Scan on idx_t_big_name (cost=0.00..4.44 rows=1 ...)
5	Index Cond: (name = 'adam1'::text)
6	-> Bitmap Index Scan on idx_t_big_name (cost=0.00..4.44 rows=1 ...)
7	Index Cond: (name = 'linda1'::text)

On the contrary the system is now using the index to search for matching rows. It is using two sub nodes then combining the results and doing a recheck.

PostgreSQL will use indexes when they make sense to the query, even if indexes exist.

19.27 Using Organized Vs Random Data

```
SELECT * FROM t_big ORDER BY id LIMIT 10;
```

	id integer	name text
1	1	adam
2	2	adam
3	3	adam
4	4	adam
5	5	adam
6	6	adam
7	7	adam
8	8	adam
9	9	adam
10	10	adam

This data is very organised and hierarchical in nature. For example the 5th row ID is 5 so it is a fair assumption that the row before would be minus 1.

```
EXPLAIN (ANALYZE true, buffers true, timing true)  
SELECT * FROM t_big WHERE id < 10000;
```

	QUERY PLAN text
1	Index Scan using idx_t_big_id on t_big (cost=0.43..404.43 rows=11714 width=9) (actual time=0.018..1.731 rows=9999 loops=1)
2	Index Cond: (id < 10000)
3	Buffers: shared hit=4 read=81
4	Planning:
5	Buffers: shared hit=4
6	Planning Time: 0.857 ms
7	Execution Time: 2.008 ms

The buffers will tell us how many times the blocks were touched, in this case 81 times. A Shared hit means it was coming from a shared cache.

```
CREATE TABLE t_big_random AS SELECT * FROM t_big ORDER BY RANDOM();  
SELECT 4000000  
Query returned successfully in 8 secs 434 msec.
```

The above will copy the rows from t-big into the new table t_big_random in a random unordered fashion.

```
SELECT * FROM t_big_random LIMIT 10;
```

	id integer	name text
1	56739	adam
2	62606	adam
3	380442	adam
4	1064236	adam
5	508516	adam
6	3991360	linda
7	1806625	adam
8	2890155	linda
9	2602731	linda
10	1026769	adam

```
CREATE INDEX idx_t_big_random_id ON t_big_random (id);  
CREATE INDEX  
Query returned successfully in 3 secs 301 msec.
```

```
VACUUM ANALYZE t_big_random;  
VACUUM  
Query returned successfully in 250 msec.
```

The VACUUM command cleans up dead tuples (obsolete rows) in PostgreSQL tables. This helps reclaim storage space, prevent transaction ID wraparound issues, and improve database performance.

```
EXPLAIN (ANALYZE true, buffers true, timing true)  
SELECT * FROM t_big_random WHERE id < 10000;
```

	QUERY PLAN text
1	Bitmap Heap Scan on t_big_random (cost=195.94..18087.15 rows=10259 width=9) (actual time=4.305..65.706 rows=9999 loops=1)
2	Recheck Cond: (id < 10000)
3	Heap Blocks: exact=8049
4	Buffers: shared hit=968 read=7111
5	-> Bitmap Index Scan on idx_t_big_random_id (cost=0.00..193.37 rows=10259 width=0) (actual time=2.828..2.828 rows=9999 loops=1)
6	Index Cond: (id < 10000)
7	Buffers: shared hit=3 read=27
8	Planning:
9	Buffers: shared hit=14 read=3
10	Planning Time: 0.309 ms
11	Execution Time: 66.293 ms

```
EXPLAIN (ANALYZE true, buffers true, timing true)
SELECT * FROM t_big_random WHERE id < 10000;
```

Same query run again to see if there is any caching.

	QUERY PLAN text	
1	Bitmap Heap Scan on t_big_random (cost=195.94..18087.15 rows=10259 width=9) (actual time=3.925..10.117 rows=9999 loops=1)	
2	Recheck Cond: (id < 10000)	
3	Heap Blocks: exact=8049	
4	Buffers: shared hit=8079	
5	-> Bitmap Index Scan on idx_t_big_random_id (cost=0.00..193.37 rows=10259 width=0) (actual time=2.470..2.471 rows=9999 loop...)	
6	Index Cond: (id < 10000)	
7	Buffers: shared hit=30	
8	Planning:	
9	Buffers: shared hit=4	
10	Planning Time: 0.281 ms	
11	Execution Time: 10.782 ms	

8049 Blocks with a runtime of 66.293ms when the query was run the first time.
 8049 Blocks with a run time of 10.782ms when the query was run the second time.
 Notice the shared hits is 30 on the second query indicating that cache was used.

```
SELECT
tablename,
attname,
correlation
FROM pg_stats
WHERE tablename IN('t_big','t_big_random')
ORDER BY 1,2;
```

	tablename name	attname name	correlation real
1	t_big	id	1
2	t_big	name	1
3	t_big_random	id	-0.0018285848
4	t_big_random	name	0.49815592

Note that in t_big ID column the correlation is 1 which means that the following row is likely to be plus 1.

Compare that with t_big_random and the correlation value is small meaning that there is a low low probability that the next row is plus 1, indicating a high level of randomness in the data.

The first time the query was run the data was disorganised without tight packing on the disk so a lot of blocks needed to be read to get the result resulting in poor query performance. The query times depend on how the data is organised in the tables and organised on the physical disk so try to build tables in an ordered manner to have optimized fast queries.

19.28 Try to Use Index Only Scan

```
EXPLAIN ANALYZE SELECT * FROM T_big WHERE id = 123456;
```

	QUERY PLAN text	🔒
1	Index Scan using idx_t_big_id on t_big (cost=0.43..8.45 rows=1 width=9) (actual time=0.963..0.964 rows=1 loops=...	
2	Index Cond: (id = 123456)	
3	Planning Time: 4.202 ms	
4	Execution Time: 0.980 ms	

Planning: 4.202ms
Execution: 0.980ms
Total: 5.182ms

```
EXPLAIN ANALYZE SELECT id FROM T_big WHERE id = 123456;
```

	QUERY PLAN text	🔒
1	Index Only Scan using idx_t_big_id on t_big (cost=0.43..8.45 rows=1 width=4) (actual time=0.696..0.698 rows=1 loops=...	
2	Index Cond: (id = 123456)	
3	Heap Fetches: 0	
4	Planning Time: 0.126 ms	
5	Execution Time: 0.732 ms	

Planning: 0.126ms
Execution: 0.732ms
Total: 0.858ms

Compare the above two queries and compare their time to run. Note that the second query has a significantly shorter execution time. The first query is an index scan so it is finding the row from the index then referencing the table to get all the data. The second query is referencing the index but as it only needs the ID field it only needs to reference the index because the id column is already indexed. **An index only scan will perform significantly better than an index scan.**

19.29 Partial Indexes

CREATE INDEX index_name **ON** table_name
WHERE conditions;

A partial index is used to improve the performance of an index while reducing the

```
SELECT pg_size_pretty(pg_indexes_size('t_big'));
```

The size of the indexes for t_big table is 112Mb

	pg_size_pretty text
1	112 MB

```
DROP INDEX idx_t_big_name;
```

```
DROP INDEX  
Query returned successfully in 51 msec.
```

```
SELECT pg_size_pretty(pg_indexes_size('t_big'));
```

	pg_size_pretty text
1	86 MB

Without the index on the name column the size of the index drops from 112Mb to 86mb

```
CREATE INDEX idx_p_t_big_name ON t_big (name)  
WHERE name NOT IN ('adam', 'linda');  
CREATE INDEX  
Query returned successfully in 723 msec.
```

As the name column contains either 'adam' or 'linda' I can use a NOT IN clause to create an index on values that are not the expected.

```
SELECT pg_size_pretty(pg_indexes_size('t_big'));
```

This is a partial index that excludes frequent values, for example gender (male, female) could be excluded from a table of values such as M, F, X etc.

	pg_size_pretty text
1	86 MB

The table indexes are still the same size because there are no values that are not 'adam' or 'linda'

customers

General Columns Advanced Constraints Partitions Parameters Security SQL

		fax	character varying	24		☐	☐	
		is_active	character	1		☐	☐	

The 'customers' table is modified to include an 'is_active' column

```
UPDATE customers
SET is_active = 'N'
WHERE customer_id IN('ALFKI','ANATR');
UPDATE 2
Query returned successfully in 67 msec.
```

```
UPDATE customers
SET is_active = 'Y'
WHERE customer_id NOT IN('ALFKI','ANATR');
UPDATE 89
Query returned successfully in 81 msec.
```

Now I update two row values for 'is_active' and set the rest to 'Y' and run a query without an index I get a total execution time of 0.171ms.

Planning: 0.091ms
Execution: 0.080ms
Total: 0.171ms

```
EXPLAIN ANALYZE SELECT * FROM customers WHERE is_active = 'N';
```

```
CREATE INDEX idx_p_customers_is_inactive ON customers (is_active)
WHERE is_active = 'N';
CREATE INDEX
Query returned successfully in 72 msec.
```

```
SELECT pg_size_pretty(pg_indexes_size('customers'));
```

```
EXPLAIN ANALYZE SELECT * FROM customers WHERE is_active = 'N';
```

I create a partial index for only the 'N' values of the 'is_active' column and I get a 32Kb total index size.

The query is using a sequential scan but if the number of rows in the table were a lot greater a partial index would be better because it would use an index scan and reduce the query time.

Planning: 0.199ms
Execution: 0.045ms
Total: 0.2.44ms

19.30 Expression Index

Is an index created on an expression e.g UPPER(column_name) or COS(column_name) or EXTRACT(day from date_column) etc.

PostgreSQL will consider using that index when the expression that defines the index appears in the WHERE clause or in the ORDER BY clause of the SQL statement.

PostgreSQL has to evaluate the expression for EACH ROW when it is INSERTED or UPDATED and use the result for indexing.

These are very expensive indexes and should only be used where necessary.

```
CREATE INDEX index_name ON table_name(expression);
```

```
CREATE table t_dates AS
SELECT d, REPEAT(md5(d::text), 10) AS padding
FROM generate_series(
TIMESTAMP '1800-01-01',
TIMESTAMP '2100-01-01',
INTERVAL '1 day') s(d);
SELECT 109574
Query returned successfully in 882 msec.
```

To test expression indexes I have created a dummy table with a column of timestamps and a column of random text data (md5 hashes). The table has a row for each day from 1800 to 2100 (100,000 rows approx)

[illegible]

```
VACUUM ANALYZE t_dates;  
VACUUM Query  
returned successfully in 589 msec.
```

```
SELECT COUNT(*) FROM t_dates;
```

	count bigint
1	109574

I run a vacuum statement on the table to create a statistic for the newly created table to help the optimizer as much as possible when calculating the best query execution plan.

```
EXPLAIN ANALYZE SELECT * FROM t_dates WHERE d BETWEEN '2001-01-01' AND '2001-01-31';
```

	QUERY PLAN text
1	Seq Scan on t_dates (cost=0.00..6624.61 rows=32 width=332) (actual time=11.345..18.258 rows=31 loops=1)
2	Filter: ((d >= '2001-01-01 00:00:00'::timestamp without time zone) AND (d <= '2001-01-31 00:00:00'::timestamp without time zon...
3	Rows Removed by Filter: 109543
4	Planning Time: 0.125 ms
5	Execution Time: 18.270 ms

18.270ms without any indexes

```
CREATE INDEX idx_t_dates_d ON t_dates (d);  
CREATE INDEX  
Query returned successfully in 104 msec.
```

```
EXPLAIN ANALYZE SELECT * FROM t_dates WHERE d BETWEEN '2001-01-01' AND '2001-01-31';
```

	QUERY PLAN text
1	Index Scan using idx_t_dates_d on t_dates (cost=0.42..10.06 rows=32 width=332) (actual time=0.035..0.042 rows=31 loops=1)
2	Index Cond: ((d >= '2001-01-01 00:00:00'::timestamp without time zone) AND (d <= '2001-01-31 00:00:00'::timestamp without time zon...
3	Planning Time: 0.912 ms
4	Execution Time: 0.056 ms

0.056ms

```
EXPLAIN ANALYZE SELECT * FROM t_dates WHERE EXTRACT(day FROM d) = 1;
```

	QUERY PLAN
	text
1	Seq Scan on t_dates (cost=0.00..6624.61 rows=548 width=332) (actual time=0.075..31.274 rows=3601 loops=...
2	Filter: (EXTRACT(day FROM d) = '1'::numeric)
3	Rows Removed by Filter: 105973
4	Planning Time: 0.999 ms
5	Execution Time: 31.425 ms

Although I have an index on the 'd' column PostgreSQL cannot use the index to perform an expression the extract part of the date so it falls back to a sequential scan, i.e. without using an index.

Now I have created an index that has extracted the day from each row of the the timestamp 'd' field

```
CREATE INDEX idx_expr_t_dates ON t_dates (EXTRACT(day FROM d));  
CREATE INDEX  
Query returned successfully in 191 msec.
```

```
ANALYZE t_dates;  
ANALYZE  
Query returned  
successfully in 557  
msec.
```

When adding indexes it is always prudent to run an analyze on the affected table to refresh the statistics that the query optimizer uses to ensure efficient query response times.

```
EXPLAIN ANALYZE SELECT * FROM t_dates WHERE EXTRACT(day FROM d) = 1;
```

	QUERY PLAN
	text
1	Bitmap Heap Scan on t_dates (cost=72.89..4954.03 rows=3674 width=332) (actual time=1.273..17.482 rows=3601 loops=1)
2	Recheck Cond: (EXTRACT(day FROM d) = '1'::numeric)
3	Heap Blocks: exact=3601
4	-> Bitmap Index Scan on idx_expr_t_dates (cost=0.00..71.97 rows=3674 width=0) (actual time=0.883..0.883 rows=3601 loop=1)
5	Index Cond: (EXTRACT(day FROM d) = '1'::numeric)
6	Planning Time: 0.335 ms
7	Execution Time: 17.763 ms

Now I have a better execution time that is using the expression index.

19.31 Adding Data While Indexing

Creating an index is super easy. However you cannot modify table data while an index is being created because PostgreSQL will create a share lock in the table until the index creation is complete. This may be an issue in a production environment with a large user base.

```
CREATE INDEX CONCURRENTLY idx_t_big_name2 on t_big (name);  
CREATE INDEX  
Query returned successfully in 2 secs 171 msec.
```

```
SELECT oid, relname, relpages, reltuples,  
i.indisunique, i.indisclustered, i.indisvalid,  
pg_catalog.pg_get_indexdef(i.indexrelid, 0, true)  
FROM pg_class c JOIN pg_index i ON c.oid = i.indrelid  
WHERE c.relname = 't_big';
```

The concurrently keyword will allow access to the data while the index is being built. However it does not guarantee the desired returns.

	oid oid	relname name	relpages integer	reltuples real	indisunique boolean	indisclustered boolean	indisvalid boolean	pg_get_indexdef text
1	82747	t_big	21622	4e+06	false	false	true	CREATE INDEX idx_t_big_id ON t_big USING btree (id)
2	82747	t_big	21622	4e+06	false	false	true	CREATE INDEX idx_p_t_big_name ON t_big USING btree (name) WHERE name <> ALL (ARRAY['adam'::text, 'linda'::te
3	82747	t_big	21622	4e+06	false	false	true	CREATE INDEX idx_t_big_name2 ON t_big USING btree (name)

The above statement will produce a table of relevant health information about an index. Interesting column is 'inisvalid' which returns true if the index is valid.

19.32 Invalidating an Index

```
SELECT oid, relname, relpages, reltuples,  
i.indisunique, i.indisclustered, i.indisvalid,  
pg_catalog.pg_get_indexdef(i.indexrelid, 0, true)  
FROM pg_class c JOIN pg_index i ON c.oid = i.indrelid  
WHERE c.relname = 'orders';
```

Invalidating an index is a way to tell PostgreSQL to not use the index without having to actually delete the index.

	oid oid	relname name	relpages integer	reltuples real	indisunique boolean	indisclustered boolean	indisvalid boolean	pg_get_indexdef text
1	16697	orders	14	830	true	false	true	CREATE UNIQUE INDEX pk_orders ON orders USING btree (order_id)
2	16697	orders	14	830	false	false	true	CREATE INDEX idx_orders_ship_city ON orders USING btree (ship_city)
3	16697	orders	14	830	false	false	true	CREATE INDEX idx_orders_customer_id_order_id ON orders USING btree (customer_id, order_id)
4	16697	orders	14	830	true	false	true	CREATE UNIQUE INDEX idx_u_orders_order_id_customer_id ON orders USING btree (order_id, customer_id)
5	16697	orders	14	830	false	false	true	CREATE INDEX idx_orders_order_date ON orders USING hash (order_date)

Invalidating an index is useful for testing on a production server when you want to force PostgreSQL to take a different path, avoiding an index. It can be used as a temporary measure to analyse indexes and performance.

To invalidate an index we need to modify a particular table called `pg_index` and set the `indisvalid` field to false.

```
EXPLAIN SELECT * FROM orders WHERE ship_country = 'USA';
```

	QUERY PLAN text
1	Seq Scan on orders (cost=0.00..24.38 rows=122 width=90)
2	Filter: ((ship_country)::text = 'USA'::text)

Here it is using a sequential scan because there is no index on the column 'ship_country' and it is an equality query.

```
CREATE INDEX idx_orders_ship_country ON orders (ship_country);
CREATE INDEX
Query returned successfully in 52 msec.
```

```
EXPLAIN SELECT * FROM orders WHERE ship_country = 'USA';
```

QUERY PLAN		
	text	
1	Bitmap Heap Scan on orders (cost=5.10..20.62 rows=122 width=90)	
2	Recheck Cond: ((ship_country)::text = 'USA'::text)	
3	-> Bitmap Index Scan on idx_orders_ship_country (cost=0.00..5.07 rows=122 width=0)	
4	Index Cond: ((ship_country)::text = 'USA'::text)	

So now the same query is using the index. But now I want to disable this index to see if the query optimizer will chose another path to the data.

```
UPDATE pg_index
SET indisvalid = false
WHERE indexrelid = (SELECT oid FROM pg_class
                     WHERE relkind = 'i'
                     AND relname = 'idx_orders_ship_country');
UPDATE 1
Query returned successfully in 72 msec.
```

This update query will change an index 'indisvalid' status from true to false.

```
SELECT oid, relname, relpages, reltuples,
i.indisunique, i.indisclustered, i.indisvalid,
pg_catalog.pg_get_indexdef(i.indexrelid, 0, true)
FROM pg_class c JOIN pg_index i ON c.oid = i.indrelid
WHERE c.relname = 'orders';
```

	oid oid	relname name	relpages integer	reltuples real	indisunique boolean	indisclustered boolean	indisvalid boolean	pg_get_indexdef text
1	16697	orders	14	830	true	false	true	CREATE UNIQUE INDEX pk_orders ON orders USING btree (order_id)
2	16697	orders	14	830	false	false	true	CREATE INDEX idx_orders_ship_city ON orders USING btree (ship_city)
3	16697	orders	14	830	false	false	true	CREATE INDEX idx_orders_customer_id_order_id ON orders USING btree (customer_id, order_id)
4	16697	orders	14	830	true	false	true	CREATE UNIQUE INDEX idx_u_orders_order_id_customer_id ON orders USING btree (order_id, customer_id)
5	16697	orders	14	830	false	false	true	CREATE INDEX idx_orders_order_date ON orders USING hash (order_date)
6	16697	orders	14	830	false	false	false	CREATE INDEX idx_orders_ship_country ON orders USING btree (ship_country)

```
EXPLAIN SELECT * FROM orders WHERE ship_country = 'USA';
```

	QUERY PLAN text
1	Seq Scan on orders (cost=0.00..24.38 rows=122 width=90)
2	Filter: ((ship_country)::text = 'USA'::text)

Now it is using a sequential scan instead of the invalidated index.

```
UPDATE pg_index
SET indisvalid = true
WHERE indexrelid = (SELECT oid FROM pg_class
                     WHERE relkind = 'i'
                     AND relname = 'idx_orders_ship_country');

UPDATE 1
Query returned successfully in 53 msec.
```

This query will activate an index so that the query optimizer will now consider the index when formulating a query plan.

19.33 Rebuilding an Index

AN index is a separate file within the database to the table data and sometimes an index may get corrupted or the data in an index may become desynchronised with the table. Although this is an unlikely occurrence it can sometimes be useful to rebuild an index if his should occur.

`REINDEX [ (VERBOSE) ] { INDEX | TABLE | SCHEMA | DATABASE | SYSTEM } [ CONCURRENTLY ] name`

```
REINDEX INDEX idx_orders_customer_id_order_id;  
REINDEX  
Query returned successfully in 42 msec.
```

I can re-index a specific index by it's name

```
REINDEX (VERBOSE) INDEX idx_orders_customer_id_order_id;  
INFO: index "idx_orders_customer_id_order_id" was reindexed  
REINDEX  
Query returned successfully in 37 msec.
```

VERBOSE will give a little more output info on a reindex.

```
REINDEX (VERBOSE) TABLE orders;  
INFO: index "pg_toast_16697_index" was reindexed  
INFO: index "pk_orders" was reindexed  
INFO: index "idx_orders_ship_city" was reindexed  
INFO: index "idx_orders_customer_id_order_id" was reindexed  
INFO: index "idx_u_orders_order_id_customer_id" was reindexed  
INFO: index "idx_orders_order_date" was reindexed  
INFO: index "idx_orders_ship_country" was reindexed  
REINDEX  
Query returned successfully in 46 msec.
```

I can reindex all indexes on a specific table by using TABLE table_name.

```
REINDEX (VERBOSE) SCHEMA public;
INFO: index "pk_usstates" was reindexed
INFO: table "public.us_states" was reindexed
....
INFO: index "idx_t_big_random_id" was reindexed
INFO: table "public.t_big_random" was reindexed
REINDEX
Query returned successfully in 7 secs 915 msec.
```

I can reindex all
indexes in a schema

```
REINDEX (VERBOSE) DATABASE northwind;
INFO: index "pk_usstates" was reindexed
INFO: table "public.us_states" was reindexed
....
INFO: index "idx_t_big_random_id" was reindexed
INFO: table "public.t_big_random" was reindexed
REINDEX
Query returned successfully in 7 secs 915 msec.
```

Or all the indexes in a
database

When running schema, database or system reindexes in a production server, it will place a lock on affected tables until the reindex is complete so use CONCURRENTLY to prevent all users from being shut out INSERT; UPDATE and DELETE while the re-index is occurring.

```
REINDEX (VERBOSE) TABLE CONCURRENTLY orders;
```

```
BEGIN
    REINDEX INDEX index_name
    REINDEX TABLE table_name
END
```

Reindex for indexes and tables can be run in a transaction block.
Note that SCHEMA, DATABASE and SYSTEM cannot be put in a transaction block.

20. All About Views

[20.1 Introduction to Views](#)

[20.2 Creating a View](#)

[20.3 Rename a View](#)

[20.4 Delete a View](#)

[20.5 Using Filters with Views](#)

[20.6 A View with Union of Multiple Tables](#)

[20.7 Connecting Multiple tables with a Single View](#)

[20.8 Rearrange Columns in a View](#)

[20.9 Delete a Column in a View](#)

[20.10 Add a Column in a View](#)

[20.11 Regular Views are Dynamic](#)

[20.12 What is an Updatable View?](#)

[20.13 Updatable View with CRUD operations](#)

[20.14 Updating Views with Check options](#)

[20.15 Updatable Views with LOCAL & CASCADE CHECK OPTION](#)

20. All About Views

[20.16 What is a Materialized View?](#)

[20.17 Creating a Materialized View](#)

[20.18 Drop materialized View](#)

[20.19 Changing Materialized View Data](#)

[20.20 How to Check if a Materialized View is Populated](#)

[20.21 Refreshing Data in a Materialized View](#)

[20.22 Why not use a Table instead of a Materialized View?](#)

[20.23 The Downsides of Using Materialized Views](#)

[20.24 Using Materialized Views for websites Page Analysis](#)

[20.25 List All Materialized Views by a SELECT statement](#)

[20.26 List all Materialized Views with no Unique Index](#)

[20.27 Quick Queries for Materialized Views](#)

20.1 Introduction to Views

One of the advantages of a programming language is that it allows us to automate repetitive, boring tasks. For example, If you have to run the same query every day or month to update the same table, sooner or later you'll search for a shortcut to accomplish the task. The good news is that shortcuts exist in PostgreSQL and it is called a 'view'.

A query can be saved into a view. So instead of writing long queries, you can just refer to a view. A view is a database object that is of a 'stored query'.

A view is a virtual table you create dynamically using a saved query acting as a 'virtual table'.

Views help to encapsulate queries and logic into reusable PostgreSQL database objects that will speed up workflow.

Views are like Don't Repeat Yourself. (DRY)

Similar to a regular table you can:

- Query a view,
- Join a view to regular tables (or other views)
- Use the view to update or insert data into the table it's based on, albeit with some caveats.

Note that a regular view does not store any data except the 'materialized views'

Views are Especially Helpful because they allow:

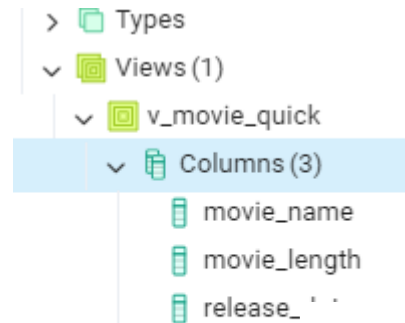
- Avoid duplicate effort by letting you write a query once and access the result when needed
- Reduce complexity for yourself and other database users by showing only columns relevant to your needs
- Provide security by limiting access to only certain columns in a table
- Like a table, you can grant permission to users through a view that contains specific data or columns that the users are authorised to see.

20.2 Creating a View

CREATE or REPLACE VIEW view_name AS query

If there is an existing view name then it will overwrite it

```
CREATE OR REPLACE VIEW v_movie_quick AS
SELECT
movie_name,
movie_length,
release_date
FROM movies mv;
CREATE VIEW
Query returned successfully in 71 msec.
```

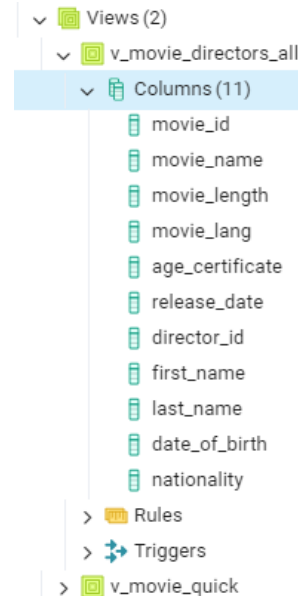


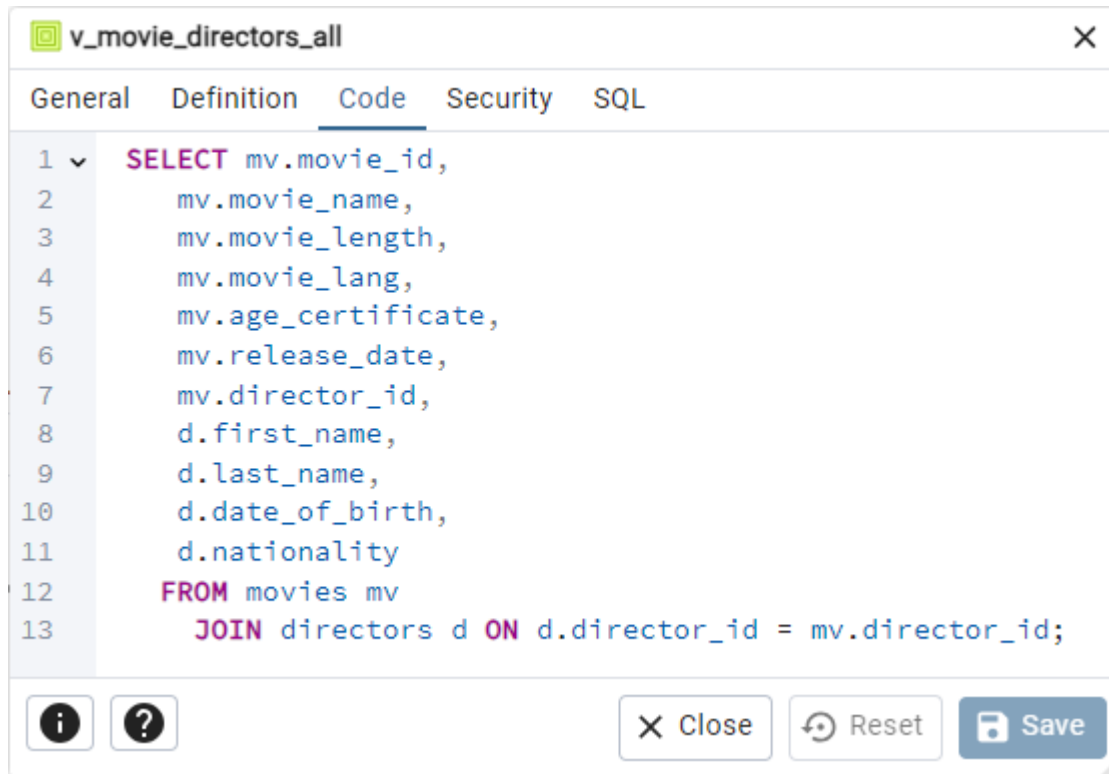
In PgAdmin4 →
database →
schema →
views

```
CREATE OR REPLACE VIEW v_movie_directors_all AS
SELECT
*
FROM movies mv
INNER JOIN directors d ON d.director_id = mv.director_id
ERROR: column "director_id" specified more than once
SQL state: 42701
```

This fails because I am specifying the same column name more than once. 'director_id' is coming once from the directors table and once again from the movies table.

```
CREATE OR REPLACE VIEW v_movie_directors_all AS
SELECT
mv.movie_id,
mv.movie_name,
mv.movie_length,
mv.movie_lang,
mv.age_certificate,
mv.release_date,
mv.director_id,
d.first_name,
d.last_name,
d.date_of_birth,
d.nationality
FROM movies mv
INNER JOIN directors d ON d.director_id = mv.director_id
```





In PgAdmin4 → database
 → schema → views →
 View_name → Right click
 → Properties

```
SELECT * FROM v_movie_quick;
```

	movie_name character varying (100)	movie_length integer	release_date date
1	A Clockwork Orange	112	1972-02-02
2	Apocalypse Now	168	1979-08-15
3	Battle Royale	111	2001-01-04
4	Blade Runner	117	1982-06-25

```
SELECT * FROM v_movie_directors_all;
```

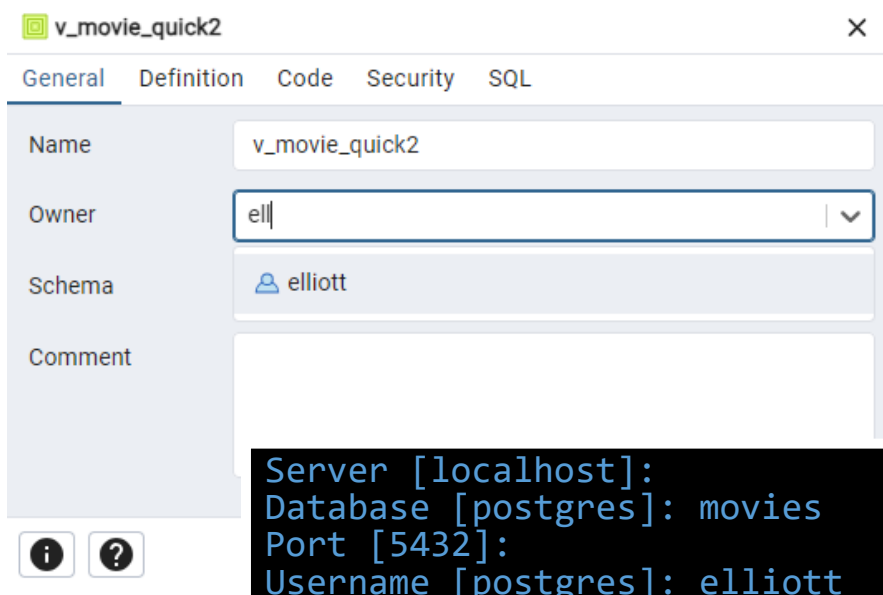
To use a view, select data from it
 like it was a normal table.

A view can be named whatever you want but a
 sensible naming convention is advised. Using
 v_view_name is a good way to distinguish a
 view from a table.

20.3 Rename a View

```
ALTER VIEW view_name RENAME TO new_viw_name;
```

```
ALTER VIEW v_movie_quick RENAME TO v_movie_quick2;  
ALTER VIEW  
Query returned successfully in 109 msec.
```



The screenshot shows the 'v_movie_quick2' view properties in the PostgreSQL Enterprise Console. The 'Name' field is 'v_movie_quick2'. The 'Owner' field is 'elliott', which is highlighted in blue. The 'Schema' field is also 'elliott'. The 'Comment' field is empty. The 'General' tab is selected, and the 'Definition', 'Code', 'Security', and 'SQL' tabs are also visible.

By default the owner of the view is postgres.

Leaving the view owner as postgres I log in via a terminal as 'elliott' and try and rename the view and it gives me an error.

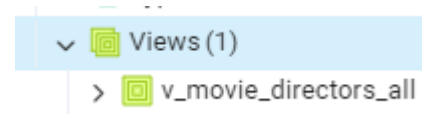
This is because only the owner and postgres super user can change the name of a view.

```
Server [localhost]:  
Database [postgres]: movies  
Port [5432]:  
Username [postgres]: elliott  
Password for user elliott:  
psql (17.2)  
WARNING: Console code page (850) differs from Windows code page (1252)  
8-bit characters might not work correctly. See psql reference  
page "Notes for Windows users" for details.  
Type "help" for help.  
  
movies=> ALTER VIEW v_movie_quick2 RENAME TO v_movie_quick;  
ERROR: relation "v_movie_quick2" does not exist  
movies=>
```


20.4 Delete a View

```
DROP VIEW view_name;
```

```
DROP VIEW v_movie_quick;  
DROP VIEW  
Query returned successfully in 40 msec.
```



View is deleted

20.5 Using Filters with Views

```
CREATE OR REPLACE VIEW v_movies_after_1997  
AS  
SELECT *  
FROM movies  
WHERE release_date > '1997-12-31'  
ORDER BY release_date ASC;  
CREATE VIEW  
Query returned successfully in 65 msec.
```

Create a view to filter for all movies released after 1987. Steps:

1. Define and test the query
2. Prepend the query in a CREATE VIEW

```
SELECT *  
FROM v_movies_after_1997  
WHERE movie_lang = 'English';
```

I can use filters on a view also in the same way as using filters on a table.

Showing rows: 1 to 15  Page No: 1							
	movie_id integer	movie_name character varying (100)	movie_length integer	movie_lang character varying (20)	age_certificate character varying (10)	release_date date	director_id integer
1	30	Rushmore	104	English	12	1998-11-12	3
2	10	Eyes Wide Shut	130	English	18	1999-07-16	13
3	12	Gladiator	165	English	15	2000-05-05	27

```
SELECT *
FROM v_movies_after_1997
WHERE movie_lang = 'English'
AND age_certificate = '12';
```

I can use filters on a view also in the same way as using filters on a table.

	movie_id integer	movie_name character varying (100)	movie_length integer	movie_lang character varying (20)	age_certificate character varying (10)	release_date date	director_id integer
1	30	Rushmore	104	English	12	1998-11-12	3
2	51	V for Vendetta	140	English	12	2006-03-17	19
3	52	Watchmen	138	English	12	2009-03-06	29

```
SELECT *
FROM v_movies_directors_all
WHERE nationality IN('American', 'Japanese');
```

Get all movies with directors from America or Japan.

Data Output Messages Notifications								
+ 📄 ▼ 📋 ▼ 🗑️ 📦 ⬇️ 📈 SQL Showing rows: 1 to 31 ✎ Page No: 1 of 1 ⏪ ⏴ ⏵ ⏩								
	g (20)	age_certificate character varying (10)	release_date date	director_id integer	first_name character varying (150)	last_name character varying (150)	date_of_birth date	nationality character varying (20)
1		18	1972-02-02	13	Stanley	Kubrick	1928-07-26	American
2		15	1979-08-15	9	Francis	Ford Coppola	1939-04-07	American
3		18	2001-01-04	10	Kinji	Fukasaku	1930-07-03	Japanese

I could write a long winded join query to get all the column data from both the movies and directors table then apply the filter in a where clause but it is much easier to just pull the information from the view and perform a simpler filter on the view table.

20.6 A View with Union of Multiple Tables

```
CREATE OR REPLACE VIEW v_all_actors_directors AS
SELECT
first_name,
last_name,
'actors' AS people_type
FROM actors
UNION ALL
SELECT
first_name,
last_name,
'directors' AS people_type
FROM directors
```

Create a view of actors and directors first and last names.

```
SELECT * FROM v_all_actors_directors ORDER BY people_type, first_name;
```

	first_name character varying (150) 🔒	last_name character varying (150) 🔒	people_type text 🔒
1	Abbie	Cornish	actors
2	Adrien	Brody	actors
3	Aki	Maeda	actors
4	Alec	Guinness	actors

```
SELECT * FROM v_all_actors_directors WHERE first_name = 'John' ORDER BY people_type;
```

	first_name character varying (150) 🔒	last_name character varying (150) 🔒	people_type text 🔒
1	John	Cleese	actors
2	John	Travolta	actors
3	John	Lasseter	directors

20.7 Connecting Multiple tables with a Single View

```
CREATE OR REPLACE VIEW v_movies_directors_revenues AS
SELECT
mv.movie_id,
mv.movie_name,
mv.movie_length,
mv.movie_lang,
mv.age_certificate,
mv.release_date,
mv.director_id,
d.first_name,
d.last_name,
d.nationality,
d.date_of_birth,
r.revenues_domestic,
r.revenues_international
FROM movies mv
INNER JOIN directors d ON d.director_id = mv.director_id
INNER JOIN movies_revenues r ON r.movie_id = mv.movie_id
CREATE VIEW
Query returned successfully in 84 msec.
```

Compiling data from multiple tables into 1 using views makes for simpler queries later on.

The view is like a good base layer to start filtering from.

```
SELECT * FROM v_movies_directors_revenues WHERE age_certificate = '12';
```

	movie_lang character varying (20)	age_certificate character varying (10)	release_date date	director_id integer	first_name character varying (150)	last_name character varying (150)	nationality character varying (20)	date_of_birth date	revenues_domestic numeric (10,2)	revenues_international numeric (10,2)
1	English	12	1998-11-12	3	Wes	Anderson	American	1969-05-01	16.90	[null]
2	English	12	1997-05-09	5	Luc	Besson	French	1959-03-18	64.10	200.30
3	English	12	1997-12-19	6	James	Cameron	American	1954-08-16	659.20	1528.10
4	Chinese	12	2000-07-06	15	Ang	Lee	Chinese	1954-10-23	129.10	95.10

20.8 Rearrange Columns in a View

```
CREATE VIEW v_directors AS
SELECT
first_name,
last_name
FROM directors;
```

	first_name character varying (150) 🔒	last_name character varying (150) 🔒
1	Tomas	Alfredson
2	Paul	Anderson
3	Wes	Anderson
4	Richard	Ayoade

```
CREATE VIEW v_directors AS
SELECT
last_name,
first_name
FROM directors;
```

The order that I select the columns in the select query is the order that the columns will appear in the table.

	last_name character varying (150) 🔒	first_name character varying (150) 🔒
1	Alfredson	Tomas
2	Anderson	Paul
3	Anderson	Wes
4	Ayoade	Richard

After deleting the view I can recreate it with the columns in the order that I want.

20.9 Delete a Column in a View

PostgreSQL does not support removing an existing column in a view

The query must generate the same columns that were generated when the view was created.

The new columns must have the same names, data types and in the same order.

To modify the columns in a view delete it and create a new one that meets the requirements.

20.10 Add a Column in a View

```
CREATE VIEW v_directors AS
SELECT
last_name,
first_name
FROM directors;
```

Previously I create a view like this with the column order of last_name, first_name.

I can add more columns as long as I maintain the order and name of the existing columns.

```
CREATE OR REPLACE VIEW v_directors AS
SELECT
last_name,
first_name,
nationality
FROM directors;
CREATE VIEW
Query returned successfully in 47 msec.
```

```
CREATE OR REPLACE VIEW v_directors AS
SELECT
nationality,
last_name,
first_name,
date_of_birth
FROM directors;
ERROR: cannot change name of view column "last_name" to "nationality"
HINT: Use ALTER VIEW ... RENAME COLUMN ... to change name of view column instead.
SQL state: 42P16
```

If I try and replace a view with one that has the columns in a different order then it will throw an error.

20.11 Regular Views are Dynamic

A regular view does not store data but is just a **query template** that runs and **provides the freshest data direct from the underlying data tables.**

20.12 What is an Updatable View?

An updatable view allows you to update the data on the underlying table data. However there are some rules to follow:

- The 'query' must have one FROM entry which can either be a table or another updatable view.
- The 'query' cannot contain the following at the top level: DISTINCT, GROUP BY, WITH, LIMIT, OFFSET, UNION, INTERSET, EXCEPT, HAVING
- The select 'Query' cannot contain any window function, set returning function, or aggregate function. SUM, COUNT, AVG, MIN, MAX

You can use the following operations to update the data:

- INSERT, UPDATE, DELETE along with a corresponding WHERE clause!

When you perform INSERT, UPDATE, DELETE operations, user must have corresponding privilege on the view, but you don't need to have the privilege on the underlying table!!!!

This will help a lot on securing the database!!!!

20.13 Updatable View with CRUD operations

```
CREATE OR REPLACE view vu_directors AS
SELECT
first_name,
last_name
FROM directors;
```

I create a view as don previously. For ease of understanding the name is vu_% to signify updatable view.

```
INSERT INTO vu_directors (first_name) VALUES ('dir1'),('dir2');
INSERT 0 2
Query returned successfully in 40 msec.
SELECT * FROM vu_directors;
```

	first_name character varying (150)	last_name character varying (150)
36	Robert	Zemeckis
37	Yimou	Zhang
38	james	David
39	dir1	[null]
40	dir2	[null]

I can insert into the view and see in the view that the two additional directors first_names are added.

I can also note that the data that was added via the view has been put into the underlying data table.

```
SELECT * FROM directors ORDER BY director_id DESC;
```

	director_id [PK] integer	first_name character varying (150)	last_name character varying (150)	date_of_birth date	nationality character varying (20)	add_date date	update_date date
1	40	dir2	[null]	[null]	[null]	[null]	[null]
2	39	dir1	[null]	[null]	[null]	[null]	[null]
3	38	james	David	2010-01-01	American	[null]	[null]


```
DELETE FROM vu_directors WHERE first_name = 'dir1';
DELETE FROM vu_directors WHERE first_name = 'dir2';
```

Deleting rows from the view removes the rows from the underlying data table.

```
SELECT * FROM directors ORDER BY director_id DESC;
```

	director_id [PK] integer	first_name character varying (150)	last_name character varying (150)	date_of_birth date	nationality character varying (20)	add_date date	update_date date
1	38	james	David	2010-01-01	American	[null]	[null]
2	37	Yimou	Zhang	1950-04-02	Chinese	[null]	[null]

20.14 Updating Views with Check options

With check option clause ensures that the changes to the base tables through the view satisfy the view-defining condition, providing more security benefits.

```
CREATE TABLE countries (
    country_id SERIAL PRIMARY KEY,
    country_code VARCHAR(4),
    city_name VARCHAR(100)
);
```

```
INSERT INTO countries (country_code, city_name) VALUES
('US', 'New Yory'),
('US', 'New Jersey'),
('UK', 'London');
```

```
SELECT * FROM countries;
```

	country_id [PK] integer	country_code character varying (4)	city_name character varying (100)
1	1	US	New Yory
2	2	US	New Jersey
3	3	UK	London

Starting point for this lab is a simple table with three columns of data.

```
CREATE OR REPLACE VIEW v_cities_us AS
SELECT
country_id,
country_code,
city_name
FROM countries
WHERE country_code = 'US';
```

A simple view is created but has a WHERE clause in it to filter rows that have 'country_code' = 'US'

Because of the where clause used in the view creation the output of the view is restricted.

```
SELECT * FROM v_cities_us;
```

	country_id integer	country_code character varying (4)	city_name character varying (100)
1	1	US	New York
2	2	US	New Jersey

```
INSERT INTO v_cities_us (country_code, city_name) VALUES
('US', 'California');
INSERT 0 1
Query returned successfully in 54 msec.
```

```
INSERT INTO v_cities_us (country_code, city_name) VALUES
('UK', 'Manchester');
INSERT 0 1
Query returned successfully in 50 msec.
```

Even though I cannot view UK cities via the view, I can still add UK cities to the base table via the view which might be an undesired feature for security reasons.

```
SELECT * FROM countries;
```

	country_id [PK] integer	country_code character varying (4)	city_name character varying (100)
1	1	US	New York
2	2	US	New Jersey
3	3	UK	London
4	4	US	California
5	5	UK	Manchester

```
SELECT * FROM v_cities_us;
```

	country_id integer	country_code character varying (4)	city_name character varying (100)
1	1	US	New York
2	2	US	New Jersey
3	4	US	California

```
CREATE OR REPLACE VIEW v_cities_us AS
SELECT
country_id,
country_code,
city_name
FROM countries
WHERE country_code = 'US'
WITH CHECK OPTION;
CREATE VIEW
Query returned successfully in 51 msec.
```

So I will replace the view to include an additional like of WITH CHECK OPTION.

Note how the with check option restricts the user from performing an insert query that does not match the conditions of the view.

```
INSERT INTO v_cities_us (country_code, city_name) VALUES
('UK', 'Leeds');
ERROR: Failing row contains (6, UK, Leeds).new row violates check option for view
"v_cities_us"
ERROR: new row violates check option for view "v_cities_us"
SQL state: 44000
Detail: Failing row contains (6, UK, Leeds).
```

```
UPDATE v_cities_us
SET country_code = 'UK'
WHERE city_name = 'California';
ERROR: Failing row contains (4, UK, California).new row violates check option for view
"v_cities_us"
ERROR: new row violates check option for view "v_cities_us"
SQL state: 44000
Detail: Failing row contains (4, UK, California).
```

```
INSERT INTO v_cities_us (country_code, city_name) VALUES
('US', 'Chicago');
INSERT 0 1
Query returned successfully in 44 msec.
```

	country_id integer	country_code character varying (4)	city_name character varying (100)
1	1	US	New Yory
2	2	US	New Jersey
3	4	US	California
4	7	US	Chicago

20.15 Updatable Views with LOCAL & CASCADE CHECK OPTION

```
CREATE OR REPLACE VIEW v_cities_c AS
SELECT
country_id,
country_code,
city_name
FROM countries
WHERE city_name LIKE 'C%';
CREATE VIEW
Query returned successfully in 57 msec.
```

```
CREATE OR REPLACE VIEW v_cities_us_c AS
SELECT
country_id,
country_code,
city_name
FROM v_cities_c
WHERE country_code = 'US'
WITH LOCAL CHECK OPTION;
CREATE VIEW
Query returned successfully in 51 msec.
```

```
INSERT INTO v_cities_us_c (country_code, city_name) VALUES
('US', 'Connecticut');
INSERT 0 1
Query returned successfully in 40 msec.
```

```
INSERT INTO v_cities_us_c (country_code, city_name) VALUES
('US', 'Los Angeles');
INSERT 0 1
Query returned successfully in 39 msec.
```

```
SELECT * FROM v_cities_c;
```

	country_id integer	country_code character varying (4)	city_name character varying (100)
1	4	US	California
2	7	US	Chicago

We can nest views within views.

Create a simple view called 'v_cities_us_c' to list all cities starting with the letter C within the US

Note that I am creating it with LOCAL CHECK OPTION.

I can insert into the view any US city name regardless of if it starts with the letter C.

```
SELECT * FROM v_cities_c;
```

	country_id integer	country_code character varying (4)	city_name character varying (100)
1	4	US	California
2	7	US	Chicago
3	9	US	Connecticut

Los Angeles is not in the view but it did get inserted in to the base table.

The Local check option only checks the local current view conditions. It does not check anywhere else.

```
CREATE OR REPLACE VIEW v_cities_us_c AS
SELECT
country_id,
country_code,
city_name
FROM v_cities_c
WHERE country_code = 'US'
WITH CASCADED CHECK OPTION;
CREATE VIEW
Query returned successfully in 48 msec.
```

```
INSERT INTO v_cities_us_c (country_code, city_name) VALUES
('US', 'Las Vegas');
ERROR: Failing row contains (11, US, Las Vegas).new row violates check option for view
"v_cities_c"
ERROR: new row violates check option for view "v_cities_c"
SQL state:
44000 Detail: Failing row contains (11, US, Las Vegas).
```

```
SELECT * FROM countries;
```

	country_id [PK] integer	country_code character varying (4)	city_name character varying (100)
1	1	US	New Yory
2	2	US	New Jersey
3	3	UK	London
4	4	US	California
5	5	UK	Manchester
6	7	US	Chicago
7	9	US	Connecticut
8	10	US	Los Angeles

The view has been replaced to change it to CASCADED CHECK OPTION.

The cascade option check clause checks the tables above which is why this insert query fails.

```
INSERT INTO v_cities_us_c (country_code, city_name) VALUES
('US', 'Clearwater');
INSERT 0 1
Query returned successfully in 69 msec.
```

```
SELECT * FROM v_cities_c;
```

	country_id integer	country_code character varying (4)	city_name character varying (100)
1	4	US	California
2	7	US	Chicago
3	9	US	Connecticut
4	12	US	Clearwater

```
SELECT * FROM countries;
```

	country_id [PK] integer	country_code character varying (4)	city_name character varying (100)
1	1	US	New Yory
2	2	US	New Jersey
3	3	UK	London
4	4	US	California
5	5	UK	Manchester
6	7	US	Chicago
7	9	US	Connecticut
8	10	US	Los Angeles
9	12	US	Clearwater

With cascaded check option goes up the views and checks all conditions in the local and higher views whereas local check option only checks that it meets the conditions of the local view.

20.16 What is a Materialized View?

A Materialized view allows you to store the results of a query physically and update the data periodically.

A materialized view caches the result of a complex expansive query and then allows you to refresh this result periodically.

A materialized view executes the query once then holds onto those results for your viewing pleasure until you refresh the data in the materialized view again.

This is opposed to a straight up regular view which does re-execute the query every time that you access the data in it. The data in a regular view is FRESH, but your paying for the freshness in execution time. The main question in materialized views versus the regular views is FRESHNESS of data versus PERFORMANCE time.

```
CREATE MATERIALIZED VIEW IF NOT EXISTS view_name AS query  
WITH [NO] DATA;
```

If you want to load data into the materialized view at the creation time you will use WITH DATA. If you do not want any data in the MatView at time of creation use WITH NO DATA.

A MatView can be used like a regular table, for example, you can add indexes or primary keys on it, it supports VACUUM and ANALYZE commands (useful when you refresh it often).

You can mess up your source table, your end user won't notice it before the refresh, as they access the Materialized view and not the actual underlying table.

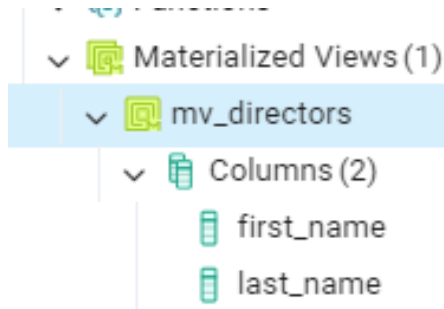
When materialized views are generally used for.

- MatViews can be used to cache results of a heavy query.
- When you need to store data that has been manipulated from its basic normalized state, and that manipulation is expensive or slow and you don't mind that the data is stale.
- That manipulation can be a complicated join or union, filtering criteria, or protected calculations etc.

20.17 Creating a Materialized View

CREATE MATERIALIZED VIEW IF NOT EXISTS view_name AS query WITH [NO] DATA;

```
CREATE MATERIALIZED VIEW IF NOT EXISTS mv_directors AS
SELECT
first_name,
last_name
FROM directors
WITH DATA
SELECT 38
Query returned successfully in 59 msec.
```



In PgAdmin4
→ database →
schema →
Materialized
Views →
mview_name

```
SELECT * FROM mv_directors;
```

The materialized view has data inside it because it was created using the WITH DATA option.

	first_name character varying (150) 🔒	last_name character varying (150) 🔒
1	Tomas	Alfredson
2	Paul	Anderson
3	Wes	Anderson
4	Richard	Avoade


```
CREATE MATERIALIZED VIEW IF NOT EXISTS mv_directors_nodata AS
SELECT
first_name,
last_name
FROM directors
WITH NO DATA
CREATE MATERIALIZED VIEW
Query returned successfully in 67 msec.
```

The materialized view has no data inside it because it was created using the WITH NO DATA option.

```
SELECT * FROM mv_directors_nodata;
ERROR: materialized view "mv_directors_nodata" has not been populated
HINT: Use the REFRESH MATERIALIZED VIEW command.
SQL state: 55000
```

When I query an empty MatView with no data inside I get an error message

The MatView is flagged as unreadable , it means that I cannot query data from the view until I load data into it.

```
REFRESH MATERIALIZED VIEW mv_directors_nodata;
REFRESH MATERIALIZED VIEW
Query returned successfully in 40 msec.
```

Now the materialized view has data inside it and I can run queries on it.

```
SELECT * FROM mv_directors_nodata;
```

	first_name character varying (150) 	last_name character varying (150) 
1	Tomas	Alfredson
2	Paul	Anderson
3	Wes	Anderson
4	Richard	Avocado

20.18 Drop Materialized View

DROP MATERIALIZED VIEW mview_name;

Do not drop a materialized view unless you truly don't want to use it anymore.

Dropping and re-creating it treats it just like a table and you lose the main benefit using a materialized view, which is that it cannot be accessed while it is updating with a simple line of code.

REFRESH MATERIALIZED VIEW CONCURRENTLY mview_name;

Anytime you run a CREATE MATERIALIZED VIEW query without the WITH NO DATA clause, you are locking the table for as long as it takes to fully populate the view.

Ideally, this would happen one time when you first set your materialized view and then you can use the CONCURRENTLY command to keep your data as many times and as accessible as possible.

20.19 Changing Materialized View Data

Should we update the data in an underlying query table or via Materialized view?

```
INSERT INTO mv_directors (first_name) VALUES ('dir1'),('dir2');  
ERROR: cannot change materialized view "mv_directors"  
SQL state: 42809
```

We cannot change the data in a materialized view.

```
INSERT INTO directors (first_name) VALUES ('dir1'),('dir2');  
INSERT 0 2  
Query returned successfully in 51 msec.
```

Step 1: update the data in the underlying table.

```
SELECT * FROM mv_directors;
```

	first_name character varying (150) 🔒	last_name character varying (150) 🔒
34	Robert	Wise
35	Kar Wai	Wong
36	Robert	Zemeckis
37	Yimou	Zhang
38	james	David

```
SELECT * FROM directors;
```

	director_id [PK] integer	first_name character varying (150) ✎	last_name character varying (150) ✎	date_of_birth date
36	36	Robert	Zemeckis	1952-05-18
37	37	Yimou	Zhang	1950-04-07
38	38	james	David	2010-01-01
39	41	dir1	[null]	[null]
40	42	dir2	[null]	[null]

Note that the data is not showing in the materialized view but is showing in the underlying table.

```
REFRESH MATERIALIZED VIEW mv_directors;  
REFRESH MATERIALIZED VIEW  
Query returned successfully in 80 msec.
```

Step 2: Refresh the materialized view to load the latest table data.

```
SELECT * FROM mv_directors;
```

	first_name character varying (150) 🔒	last_name character varying (150) 🔒
37	Yimou	Zhang
38	james	David
39	dir1	[null]
40	dir2	[null]

```
DELETE FROM mv_directors WHERE first_name = 'dir1';  
ERROR: cannot change materialized view  
"mv_directors" SQL state: 42809
```

Cannot delete or update from materialized views.

```
UPDATE mv_directors SET first_name = 'dir11' WHERE first_name = 'dir1';  
ERROR: cannot change materialized view "mv_directors"  
SQL state: 42809
```

All changes have to be made in the underlying data table then refreshed to update the contents of the materialized view. Materialized view is cached data and needs to be refreshed if any changes are made the the base data.

Some housekeeping cleanup!

```
DELETE FROM directors WHERE director_id IN(41,42);  
REFRESH MATERIALIZED VIEW mv_directors;
```

20.20 How to Check if a Materialized View is Populated

```
CREATE MATERIALIZED VIEW IF NOT EXISTS mv_directors_nodata2 AS  
SELECT  
first_name  
FROM directors  
WITH NO DATA
```

```
SELECT * FROM mv_directors_nodata2;  
ERROR: materialized view "mv_directors_nodata2" has not been populated  
HINT: Use the REFRESH MATERIALIZED VIEW command.  
SQL state: 55000
```

```
SELECT relispopulated FROM pg_class WHERE relname = 'mv_directors_nodata2';
```

The above statement will tell us if the materialized view is populated.

	relispopulated boolean	
1	false	

This could be useful if we are using external scripts to automatically trigger a refresh based on empty materialised views.

20.21 Refreshing Data in a Materialized View

Note that PostgreSQL locks the entire underlying data table when you refresh a materialized view so you cannot query against it until the refresh is completed.

```
CREATE MATERIALIZED VIEW mv_directors_us AS
SELECT
director_id,
first_name,
last_name,
date_of_birth,
nationality
FROM directors
WHERE nationality = 'American'
WITH NO DATA
```

```
SELECT relispopulated FROM pg_class WHERE
relname = 'mv_directors_us';
```

	relispopulated boolean
1	false

```
REFRESH MATERIALIZED VIEW mv_directors_us;
```

```
SELECT * FROM
mv_directors_us;
```

	director_id integer	first_name character varying (150)	last_name character varying (150)	date_of_birth date	nationality character varying (20)
1	2	Paul	Anderson	1970-06-26	American
2	3	Wes	Anderson	1969-05-01	American

CONCURRENTLY allows for the update of the materialized view without locking the underlying data table. PostgreSQL creates a temporary updated version of the materialized view, compares the two versions and performs INSERT and UPDATE on the different rows. I can query against the materialized view while it is being updated. The one requirement is that the materialized view must have a UNIQUE key.

```
REFRESH MATERIALIZED VIEW CONCURRENTLY mv_directors_us;
ERROR: cannot refresh materialized view "public.mv_directors_us" concurrently
HINT: Create a unique index with no WHERE clause on one or more columns of the
materialized view. SQL state: 55000
```

```
CREATE UNIQUE INDEX idx_u_mv_directors_us_director_id ON mv_directors_us (director_id);  
CREATE INDEX  
Query returned successfully in 56 msec.
```

```
REFRESH MATERIALIZED VIEW CONCURRENTLY mv_directors_us;  
REFRESH MATERIALIZED VIEW  
Query returned successfully in 52 msec.
```

20.22 Why not use Table instead of a Materialized View?

You can use a real table for the purpose of materialized views. The materialized view has one major benefit over the table. – The ability to refresh it without locking everyone else out.

```
Step #1: CREATE TABLE temp_table LIKE my_data;  
Step #2: INSERT INTO temp_table VALUES ();  
Step #3: DROP TABLE my_data;  
Step #3: ALTER TABLE temp_table RENAME TO my_data;
```

To update data in a table without locking users out, I would have to follow the four steps to create a temp table similar in structure to the original table then insert all the rows from the original table into the temp table and make any necessary changes. I can then delete the original table and rename to temp table to the name of the original table.

This is a lot of messy steps.

```
Step #1: CREATE MATERIALIZED VIEW mv_my_data as
        -- query here
        WITH DATA;

Step #2: REFRESH MATERIALIZED VIEW CONCURRENTLY mv_my_data;
```

CONCURRENTLY allows for update of the materialized view without locking it.

With CONCURRENTLY option, PostgreSQL creates a temporary updated version of the materialized view, compares two versions and performs INSERT and UPDATE on the differences.

You can query against the materialized view while it is being updated.

One requirement for using CONCURRENTLY is that the materialized view must have a unique index.

At the end of the day, it is all about how best we design our database structure and final desired output to our users, as materialised views are quite powerful in what they are supposed to do.

MATERIALIZED VIEWS START WITH 'M' so they must be managed!

20.23 The Downsides of Using Materialized Views

Materialized views are dependant on their base table and create dependencies on the table they reference.

If you tear down base tables and rebuild with say: adding/deleting/updating new columns, adding/updating/deleting indexes or any frequency.

A MATERIALIZED VIEW BECOMES A DEPENDENCY, THAT MUST BE MANAGED

In a database with lots of tables and serves data to multiple applications, and teams, this may not be feasible as it would require every application and team to understand how to manage dependencies that may not even improve their application.

In this case a table might be a better option, but it should be on a case by case basis.

20.24 Using Materialized Views for websites Page Analysis

```
CREATE TABLE page_clicks (  
  rec_id SERIAL PRIMARY KEY,  
  page VARCHAR(100),  
  click_time TIMESTAMP,  
  user_id BIGINT  
);
```

```
SELECT * FROM page_clicks;
```

The above generates a table populated with 10,000 rows of fake page click sample data.

I can create a materialized view that will produce a report on the total page clicks for each page.

```
CREATE MATERIALIZED VIEW mv_page_clicks AS  
SELECT  
  DATE_TRUNC('day', click_time) AS day,  
  page,  
  COUNT(*) AS total_clicks  
FROM page_clicks  
GROUP BY day, page;  
SELECT 3  
Query returned successfully in 53 msec.
```

```
INSERT INTO page_clicks (page,click_time,user_id)  
SELECT (  
  CASE (RANDOM() * 2)::INT  
    WHEN 0 THEN 'klickanalytics.com'  
    WHEN 1 THEN 'clickapis.com'  
    WHEN 2 THEN 'google.com'  
  ) AS page,  
  NOW() AS click_time,  
  (FLOOR(RANDOM() * (11111111-1000000 +1 ) +  
  1000000))::INT AS user_id  
FROM GENERATE_SERIES(1, 10000) seq;
```

Showing rows: 1 to 1000  Page No: 1 of 10    

	rec_id [PK] integer 	page character varying (100) 	click_time timestamp without time zone 	user_id bigint 
1	1	clickapis.com	2025-02-04 07:45:36.982402	50757159
2	2	klickanalytics.com	2025-02-04 07:45:36.982402	100058918
3	3	clickapis.com	2025-02-04 07:45:36.982402	37773521
4	4	klickanalytics.com	2025-02-04 07:45:36.982402	10650044

```
SELECT * FROM mv_page_clicks;
```

	day timestamp without time zone	page character varying (100)	total_clicks bigint
1	2025-02-04 00:00:00	clickapis.com	5045
2	2025-02-04 00:00:00	klickanalytics.com	2506
3	2025-02-04 00:00:00	google.com	2449

```
CREATE MATERIALIZED VIEW mv_page_clicks_daily AS
SELECT
click_time AS day,
page,
COUNT(*) AS cnt
FROM page_clicks
WHERE click_time >= DATE_TRUNC('day', NOW())
AND click_time < TIMESTAMP 'tomorrow'
GROUP BY day,page;
SELECT 3
Query returned successfully in 55 msec.
```

Now lets imagine that many days pass, the table will become very large so it will be better to split the view into daily materialized view.

Now I need to create a unique index on the materialized view so that I can concurrently refresh the data.

```
CREATE UNIQUE INDEX idx_u_mv_page_clicks_daily_day_page ON mv_page_clicks_daily (day,page);
CREATE INDEX
Query returned successfully in 48 msec.
```

```
REFRESH MATERIALIZED VIEW CONCURRENTLY mv_page_clicks_daily;
REFRESH MATERIALIZED VIEW
Query returned successfully in 42 msec.
```

20.25 List All Materialized Views by a SELECT statement

```
SELECT oid::regclass::TEXT
FROM pg_class
WHERE relkind = 'm';
```

	oid text	
1	mv_directors_nodata	
2	mv_directors_nodata2	
3	mv_directors	
4	mv_directors_us	
5	mv_page_clicks	
6	mv_page_clicks_daily	

This query will produce a table of all the materialized views in a database.

20.26 List all Materialized Views with no Unique Index

```
REFRESH MATERIALIZED VIEW CONCURRENTLY mat_view_name;
```

A materialized view MUST HAVE a unique index to refresh concurrently.

The below code snippet will list all the materialized views with no unique key created.

```
WITH matviews_with_no_unique_keys AS (  
    SELECT c.oid, c.relname, c2.relname AS idx_name  
    FROM pg_catalog.pg_class c, pg_catalog.pg_class c2, pg_catalog.pg_index i  
    LEFT JOIN pg_catalog.pg_constraint con  
    ON (  
        conrelid = i.indrelid AND conindid = i.indexrelid AND contype IN  
        ('p','u')  
    )  
    WHERE  
        c.relkind = 'm'  
        AND c.oid = i.indrelid  
        AND i.indrelid = c2.oid  
        AND indisunique  
)  
SELECT c.relname AS materialized_view_name  
FROM pg_class c  
WHERE c.relkind = 'm'  
EXCEPT  
SELECT mvk.relname  
FROM matviews_with_no_unique_keys AS mvk;
```

	materialized_view_name name
1	mv_page_clicks
2	mv_directors_nodata2
3	mv_directors
4	mv_directors_nodata

20.27 Quick Queries for Materialized Views

```
SELECT count(*) > 0 FROM pg_catalog.pg_class c
JOIN pg_namespace n ON n.oid = c.relnamespace
WHERE
c.relkind = 'm'
AND n.nspname = 'some_schema'
AND c.relname = 'some_mat_view';
```

Query if a materialized view exists

```
SELECT view_definition
FROM information_schema.views
WHERE
table_schema = 'information_schema'
AND table_name = 'views';
```

Query if a materialized view exists

```
select * from pg_matviews;
```

To list all materialized Views

```
select * from pg_matviews where matviewname = 'mv_view_name';
```

View data about an MV

21. SubQueries

[21.1 Introduction to Sub Queries](#)

[21.2 Subqueries with WHERE Clause](#)

[21.3 Subqueries with IN operator](#)

[21.4 Subqueries with JOIN](#)

[21.5 Get total revenues for all English Films](#)

[21.6 Order Entries in UNION without using ORDER BY](#)

[21.7 Subqueries with an alias](#)

[21.8 A Select without a FROM](#)

[21.9 Correlated Queries](#)

[21.10 SELECT.. IN\(subquery\)](#)

[21.11 Using ANY with Subquery](#)

[21.12 Using ALL with Subquery](#)

[21.13 Subquery Using EXISTS](#)

21.1 Introduction to Sub Queries

In a practical database structure, the data is scattered across multiple tables. Most queries that you want to make need to pull data from two or more tables.

- One way to do this is to use a JOIN operator or one of the other operational operators like UNION, INTERSECT, or EXCEPT etc. The relational operators take information from multiple tables and combine them into a single result set.
- Another way to pull data from multiple tables is to use a **nested query**. In SQL, a nested query is one in which an enclosing statement contains within it a **subquery**.
- **Subqueries are invariably SELECT statements**, but the outermost enclosing statement may also be an INSERT, UPDATE, or DELETE statement.
- A subquery can operate on a table other than the table that its enclosing statement operates on, so nested queries give you another way to extract information from multiple tables.
- A subquery is simply one query nested inside of another query, usually in the SELECT, FROM or WHERE clause.
- Subqueries are useful when the query we want to create requires a few additional steps or calculations to produce the dataset we desire. For example, subqueries are very helpful for scenarios where we want to view or compare a query by a condition that requires its own query to calculate. Instead of writing one query, then copying the results into the next query, we can use a subquery that performs both operations at once.
- Subqueries also provide us with another method of simultaneously accessing data from more than one table.
- Subqueries can help us perform a calculation in one table and then use that calculation in conjunction with another table.

21.2 Subqueries with WHERE Clause

A sub query allows you to construct a complex query.

A sub query is nested inside of another query.

Typically, it is used for a calculation or logical test that provides a value or set of data to be passed into the main portion of the query.

The syntax is pretty simple. We just enclose the subquery in () parenthesis and use it where needed.

Subqueries can be nested inside a SELECT, INSERT, UPDATE or DELETE

Subqueries can also be used after FROM or WHERE.

OUTER QUERY
(INNER QUERY)

```
SELECT
    column_list
FROM table_name
WHERE
(
    SELECT
        column_list
    FROM table_name
)
```

```
SELECT
movie_name,
movie_length
FROM movies
WHERE
movie_length >
(
SELECT
AVG(movie_length)
FROM movies
)
ORDER BY movie_length DESC;
```

	movie_name character varying (100)	movie_length integer
1	There Will Be Blood	168
2	Apocalypse Now	168
3	The Lives of Others	165
4	Gladiator	165
5	Star Wars: Empire Strikes Back	150
6	The Fifth Element	149
7	Goodfellas	148
8	City of God	145

Find average movie length and use the result in a select to find movies which are more than average movie length. Sort results by movie length.


```

SELECT
movie_name,
movie_length
FROM movies
WHERE
movie_length >
(
SELECT
AVG(movie_length)
FROM movies
WHERE movie_lang = 'English'
)
ORDER BY movie_length DESC;

```

	movie_name character varying (100)	movie_length integer
9	Titanic	143
10	V for Vendetta	140
11	City of Men	140
12	Crouching Tiger Hidden Dragon	139
13	Star Wars: Return of the Jedi	139
14	Watchmen	138
15	Pulp Fiction	136
16	Three Billboards Outside Ebbing,...	134
17	Jaws	124

The sub query is finding the average movie length of all English language films.

This avg length value is then being used by the outer query to find all movies longer than this avg length.

```

SELECT
movie_name,
movie_length
FROM movies
WHERE
movie_length >
(
SELECT
AVG(movie_length)
FROM movies
WHERE movie_lang = 'English'
)
AND movie_lang = 'English'
ORDER BY movie_length DESC;

```

	movie_name character varying (100)	movie_length integer
8	V for Vendetta	140
9	Star Wars: Return of the Jedi	139
10	Watchmen	138
11	Pulp Fiction	136
12	Three Billboards Outside Ebbs...	134
13	Jaws	134
14	Raging Bull	132
15	Eyes Wide Shut	130
16	Life of Pi	129

The sub query is finding the average movie length of all English language films.

This avg length value is then being used by the outer query to find all movies longer than this avg length.

Finally, I am filtering this to English language movies only.

```

SELECT
first_name,
last_name,
date_of_birth
FROM actors
WHERE date_of_birth <
(
SELECT
date_of_birth
FROM actors
WHERE first_name = 'Douglas'
AND last_name = 'Silva'
)
ORDER BY date_of_birth;

```

	first_name character varying (150)	last_name character varying (150)	date_of_birth date
127	Andrew	Garfield	1983-08-20
128	Paul	Dano	1984-06-19
129	Keira	Knightley	1985-03-26
130	Carey	Mulligan	1985-05-28
131	Aki	Maeda	1985-07-11
132	Mason	Gamble	1986-01-16
133	Rumi	Hiiragi	1987-08-01
134	Miyu	Irino	1988-02-19
135	Darlan	Cunha	1988-07-26

Get the first and last name of all actors who are younger than Douglas Silva (born 1988-09-27)

21.3 Subqueries with IN operator

What if an inner-query (subquery) is returning multiple records?

```
SELECT
movie_name,
movie_lang
FROM movies
WHERE movie_id IN
(
SELECT
movie_id
FROM movies_revenues
WHERE revenues_domestic >
200
);
```

	movie_name character varying (100)	movie_lang character varying (20)
1	Forrest Gump	English
2	Jaws	English
3	Spider-Man	English
4	Spider-Man 2	English
5	Spider-Man 3	English
6	Star Wars: A New Hope	English
7	Star Wars: Empire Strikes Back	English

Find all movies
where domestic
revenues are
greater than 200

```
SELECT
movie_name,
movie_lang
FROM movies
WHERE movie_id IN
(
SELECT
movie_id
FROM movies_revenues
WHERE revenues_domestic >
revenues_international
);
```

	movie_name character varying (100)	movie_lang character varying (20)
1	Crouching Tiger Hidden Drag...	Chinese
2	Jaws	English
3	Pulp Fiction	English
4	Star Wars: A New Hope	English
5	Star Wars: Empire Strikes Back	English
6	Star Wars: Return of the Jedi	English
7	Submarine	English
8	The Wizard of Oz	English

Find all movies
where domestic
revenues are higher
than international
revenues

21.4 Subqueries with JOIN

List all the directors where their movies made more than the average total revenues of all movies.

```
SELECT
d.director_id,
d.first_name || ' ' || d.last_name AS "director_name",
SUM(r.revenues_domestic + r.revenues_international) AS
"total_revenues"
FROM directors d
INNER JOIN movies mv ON mv.director_id = d.director_id
INNER JOIN movies_revenues r ON r.movie_id = mv.movie_id
WHERE (r.revenues_domestic + r.revenues_international) >
(
SELECT
AVG(revenues_domestic + revenues_international) AS
"avg_total_revenues"
FROM movies_revenues
)
GROUP BY d.director_id
ORDER BY director_name
```

	director_id [PK] integer	director_name text	total_revenues numeric
1	15	Ang Lee	609.00
2	17	George Lucas	1788.80
3	6	James Cameron	2187.30
4	14	John Lasseter	374.50
5	27	Ridley Scott	461.60
6	36	Robert Zemeck...	678.40
7	24	Sam Raimi	2497.70
8	31	Steven Spielberg	471.20
9	28	Tony Scott	357.40
10	8	Victor Fleming	400.60

21.5 Get total revenues for all English Films

```
SELECT
d.director_id,
SUM(COALESCE(r.revenues_domestic,0) + COALESCE(r.revenues_international,0)) AS
"total_reveneues"
FROM directors d
INNER JOIN movies mv ON mv.director_id = d.director_id
INNER JOIN movies_revenues r ON r.movie_id = mv.movie_id
WHERE
COALESCE(r.revenues_domestic,0) + COALESCE(r.revenues_international,0) >
(
SELECT
AVG(COALESCE(r.revenues_domestic,0) + COALESCE(r.revenues_international,0)) as
"avg_total_reveneues"
FROM movies_revenues r
INNER JOIN movies mv ON mv.movie_id = r.movie_id
WHERE mv.movie_lang = 'English'
)
GROUP BY d.director_id
ORDER BY 2 DESC, 1 ASC
```

	director_id [PK] integer 	total_reveneues numeric 
1	24	2497.70
2	6	2187.30
3	17	1788.80
4	36	678.40
5	15	609.00
6	31	471.20
7	27	461.60
8	8	400.60
9	14	374.50
10	28	357.40

21.6 Order Entries in UNION without using ORDER BY

```
SELECT *
FROM
(
  SELECT first_name, 0 myorder, 'actors' AS "table_name" FROM actors
  UNION
  SELECT first_name, 1, 'directors' AS "table_name" FROM directors
) t
ORDER BY myorder;
```

	first_name character varying (150) 🔒	myorder integer 🔒	table_name text 🔒
137	Maggie	0	actors
138	Tim	0	actors
139	Tatsuya	0	actors
140	Billy	0	actors
141	Sebastian	0	actors
142	Martin	1	directors
143	Paulo	1	directors
144	Richard	1	directors
145	George	1	directors
146	Kinji	1	directors

21.7 Subqueries with an alias

```
SELECT
*
FROM
(
  SELECT
  *
  FROM movies
) t1
```

If I am selecting * in a sub query and also selecting * from the outer query then I need to alias the subquery

21.8 A Select Without a FROM

```
SELECT
(
SELECT MAX(revenues_domestic) AS "max Domestic Revenues" FROM movies_revenues
),
(
SELECT MIN(revenues_domestic) AS "Min Domestic Revenues" FROM movies_revenues
),
(
SELECT AVG(revenues_domestic) AS "Avg Domestic Revenues" FROM movies_revenues
)
```

I have three sub queries within an outer queries that is only selecting the contents of the sub queries. There is no FROM argument in the main query.

	max Domestic Revenues numeric	Min Domestic Revenues numeric	Avg Domestic Revenues numeric
1	659.20	0.30	119.1562500000000000

21.9 Correlated Queries

A correlated subquery is a subquery that contains a reference to a table (in the parent query) that also appears in the outer query. PostgreSQL evaluates from inside to outside.

```
SELECT
mv1.movie_name,
mv1.movie_lang,
mv1.movie_length,
mv1.age_certificate
FROM movies mv1
WHERE mv1.movie_length >
(
SELECT
MIN(movie_length)
FROM movies mv2
WHERE mv1.age_certificate = mv2.age_certificate
)
ORDER BY mv1.movie_length ASC
```

	movie_name character varying (100)	movie_lang character varying (20)	movie_length integer	age_certificate character varying (10)
1	A Clockwork Orange	English	112	18
2	Apocalypse Now	English	168	15
3	Battle Royale	Japanese	111	18
4	Blade Runner	English	121	15
5	Chungking Express	Chinese	113	15
6	City of God	Portuguese	145	18
7	City of Men	Portuguese	140	15
8	Cold Fish	Japanese	108	18

List all movie names, movie language, movie age certification for all movies with above minimum length for each age certification.

```
SELECT
ac1.first_name,
ac1.last_name,
ac1.date_of_birth,
ac1.gender
FROM actors ac1
WHERE ac1.date_of_birth >
(
SELECT
MIN(date_of_birth)
FROM actors ac2
)
ORDER BY ac1.gender, ac1.date_of_birth;
```

	first_name character varying (150)	last_name character varying (150)	date_of_birth date	gender character (1)
45	Rumi	Hiiragi	1987-08-01	F
46	Yasmin	Paige	1991-06-24	F
47	Saoirse	Ronan	1994-04-12	F
48	Ivana	Baquero	1994-06-11	F
49	Lina	Leandersson	1995-09-27	F
50	Scatman	Crothers	1910-05-23	M
51	Alec	Guinness	1914-04-02	M
52	Marlon	Brando	1924-04-03	M
53	Dick	Van Dyke	1925-12-13	M
54	Gilbert	Bruneau	1928-01-01	M

List first name, last name and DOB for the oldest actors for each gender.

21.10 SELECT.. IN(subquery)

Find suppliers that are same countries as customers

```
SELECT
*
FROM suppliers
WHERE
country IN (SELECT country FROM customers);
```

```
SELECT column_list
FROM table_name
WHERE
Column_name IN
(SELECT statement)
```

	supplier_id [PK] smallint	company_name character varying (40)	contact_name character varying (30)	contact_title character varying (30)	address character varying (60)	city character varying (15)	region character varying (15)	postal_code character va
1	1	Exotic Liquids	Charlotte Cooper	Purchasing Manager	49 Gilbert St.	London	[null]	EC1 4SD
2	2	New Orleans Cajun ...	Shelley Burke	Order Administrator	P.O. Box 78934	New Orleans	LA	70117
3	3	Grandma Kelly's Ho...	Regina Murphy	Sales Representati...	707 Oxford Rd.	Ann Arbor	MI	48104
4	5	Cooperativa de Que...	Antonio del Valle S...	Export Administrator	Calle del Rosal 4	Oviedo	Asturias	33007
5	8	Specialty Biscuits, ...	Peter Wilson	Sales Representati...	29 King's Way	Manchester	[null]	M14 GSD
6	9	PR Knäckebröd AB	Lars Peterson	Sales Agent	Kaloadagatan 13	Göteborg	[null]	S-415 67

Find customers that are in the same city as suppliers

```
SELECT
*
FROM customers
WHERE
city IN (SELECT city FROM suppliers);
```

	customer_id [PK] character	company_name character varying (40)	contact_name character varying (30)	contact_title character varying (30)	address character varying (60)	city character varying (15)	region char
1	AROUT	Around the Horn	Thomas Hardy	Sales Representative	120 Hanover Sq.	London	[null]
2	ALFKI	Alfreds Futterkiste	Maria Anders	Sales Representative	Obere Str. 57	Berlin	[null]
3	BSREX	B's Beverages	Victoria Ashworth	Sales Representative	Fauntleroy Circus	London	[null]

21.11 Using ANY with Subquery

```
SELECT column_list
FROM table_name
WHERE
Column_name operator_expression ANY (SELECT statement)
```

```
SELECT
*
FROM customers
WHERE customer_id = ANY (
SELECT
customer_id
FROM orders o
INNER JOIN order_details od ON od.order_id =
o.order_id
WHERE
od.quantity > 20
)
```

Find customer details where they ordered more than 20 items in a single order

	customer_id [PK] character	company_name character varying (40)	contact_name character varying (30)	contact_title character varying (30)	address character varying (60)
1	ANTON	Antonio Moreno Taquería	Antonio Moreno	Owner	Mataderos 2312
2	AROUT	Around the Horn	Thomas Hardy	Sales Representative	120 Hanover Sq.
3	ALFKI	Alfreds Futterkiste	Maria Anders	Sales Representative	Obere Str. 57
4	BERGS	Berglunds snabbköp	Christina Berglund	Order Administrator	Berguvsvägen 8
5	BLAUS	Blauer See Delikatessen	Hanna Moos	Sales Representative	Forsterstr. 57
6	BLONP	Blondesddsl père et fils	Frédérique Citeaux	Marketing Manager	24, place Kléber

21.12 Using ALL with Subquery

SELECT column_list
FROM table_name
WHERE
Column_name **operator_expression** **ALL** (**SELECT** statement)

=, !=, >, <, <=, NOT

Find all products where
there order amount were
lower than the average
amount of all the products.

```
SELECT
*
FROM products
INNER JOIN order_details ON order_details.product_id = products.product_id
WHERE
((order_details.unit_price*order_details.quantity)-order_details.discount) < ALL
(
    SELECT
    AVG((unit_price*quantity)-discount)
    FROM order_details
    GROUP BY product_id
);
```

	product_id smallint	product_name character varying (40)	supplier_id smallint	category_id smallint	quantity_per_unit character varying (20)	unit_price real	units_in_stock smallint	units_on_order smallint
1	33	Geitost	15	4	500 g	2.5	112	0
2	37	Gravad lax	17	8	12 - 500 g pkgs.	26	11	50
3	33	Geitost	15	4	500 g	2.5	112	0
4	33	Geitost	15	4	500 g	2.5	112	0
5	24	Guaraná Fantástica	10	1	12 - 355 ml cans	4.5	20	0

21.13 Subquery Using EXISTS

```
SELECT column_list
FROM table_name
WHERE EXISTS (SELECT column_name FROM table_name WHERE condition)
```

Find suppliers with products that cost more than \$100

```
SELECT
*
FROM suppliers
WHERE EXISTS
(
SELECT
*
FROM products
WHERE unit_price > 100
AND products.supplier_id = suppliers.supplier_id
);
```

	supplier_id [PK] smallint	company_name character varying (40)	contact_name character varying (30)	contact_title character varying (30)	address character varying (60)	city character varying (30)
1	12	Plutzer Lebensmittelgroßmärkte AG	Martin Bein	International Marketing Mgr.	Bogenallee 51	Frankfurt
2	18	Aux joyeux ecclésiastiques	Guylène Nodier	Sales Manager	203, Rue des Francs-Bourgeois	Paris

22. Common Table Expressions (CTE)

[22.1 Introduction to CTEs](#)

[22.2 CTE query examples](#)

[22.3 Combine CTE with a Table](#)

[22.4 Get Total Revenues for Each Director](#)

[22.5 Simultaneously DELETE INSERT via CTE](#)

[22.6 Recursive CTEs](#)

[22.7 Parent-Child relationship via CTEs](#)

22.1 Introduction to CTEs

1. CTE is a temporary result taken from a SQL statement.
2. A second approach to create temporary tables for query data instead of using sub queries in a FROM clause.
3. A good alternative to sub queries.
4. Using the CTE, you can define one or more tables upfront with subqueries.
5. Unlike subqueries, CTEs can be referenced multiple times in multiple places in query statement.
6. CTEs can be used to improve readability and interpretability of the code.
7. CTEs are non-recursive type by default.
8. The lifetime of a CTE is equal to the lifetime of a query.
9. From PostgreSQL v12, two new options have been introduced for the execution of a CTE, namely:
 - MATERIALIZED Performs a CTE that materializes a temporary result set
 - NON MATERIALIZED Do not materialize a temporary result set (default option)
10. CTEs are usually used to simplify complex joins and subqueries
11. Ability to create Recursive queries
12. Use in conjunction with window functions e.g. RANK() etc.

22.2 CTE query examples

```
WITH cte_name (column_list [OPTIONAL]) AS (  
  Cte_query_definition  
)  
Statement;
```

The Statement can contain SELECT,
INSERT, UPDATE OR DELETE

Create a number series from 1 to 10

```
WITH num AS  
(  
  SELECT * FROM generate_series(1,10) AS id  
)  
  
SELECT * from num;
```

This CTE is using the function to
generate_series() and the output
is converted as a temporary table
and passed to our num

From outside the CTE I call it by
selecting the results of the CTE.

	id integer
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	10

List all movies by director_id = 1

```
WITH cte_director_1  
(  
  SELECT  
  *  
  FROM movies mv  
  INNER JOIN directors d ON d.director_id = mv.director_id  
  WHERE d.director_id = 1  
)  
  
SELECT  
*  
FROM cte_directors_1
```

Within the CTE we can have a
very complex query that pulls a
lot of data from the database.

Then we can perform more
queries on the data from the
CTE.

View all long movies over 120 mins

```
WITH cte_long_movies AS
(
  SELECT
    movie_name,
    movie_length,
    (
      CASE
        WHEN movie_length < 100 THEN 'SHORT'
        WHEN movie_length < 120 THEN 'MEDIUM'
        ELSE 'LONG'
      END
    ) AS m_length
  FROM movies
)
```

```
SELECT
*
FROM cte_long_movies
WHERE m_length = 'LONG';
```

I can then query on the CTE generated table to filter for only the long movies.

Within this CTE I am selecting 'movie_name', 'movie_length' plus defining a third column with an alias called 'm_length', inside a case statement where there are three options. This CTE produces a table like shown below where all movies have an 'm_length' value of SHORT, MEDIUM or LONG

	movie_name character varying (100)	movie_length integer	m_length text
25	Oldboy	130	LONG
26	Pans Labyrinth	98	SHORT
27	Ponyo	107	MEDIUM

	movie_name character varying (100)	movie_length integer	m_length text
1	Apocalypse Now	168	LONG
2	Blade Runner	121	LONG
3	City of God	145	LONG
4	City of Men	140	LONG
5	Crouching Tiger Hidden Dragon	139	LONG
6	Eyes Wide Shut	120	LONG

22.3 Combine CTE with a Table

Calculate total revenues for each director and combine with a table

```
WITH cte_movie_count AS
(
SELECT
d.director_id,
SUM(r.revenues_domestic + r.revenues_international) AS
total_revenues
FROM directors d
INNER JOIN movies mv ON d.director_id = mv.director_id
INNER JOIN movies_revenues r ON r.movie_id = mv.movie_id
GROUP BY d.director_id
)
```

```
SELECT
d.director_id,
d.first_name,
d.last_name,
cte.total_revenues
FROM cte_movie_count cte
INNER JOIN directors d ON d.director_id = cte.director_id
```

I can then query from the results of the CTE and join it to any existing table to produce whatever columns I want.

The CTE is producing a temporary table of columns: 'director_id' and 'total_revenues' where total revenues is the sum of domestic and international sales for each director.

	director_id [PK] integer	total_revenues numeric
1	29	185.00
2	4	0.90
3	34	[null]
4	32	[null]
5	7	84.20

	director_id [PK] integer	first_name character varying (150)	last_name character varying (150)	total_revenues numeric
1	1	Tomas	Alfredson	11.20
2	2	Paul	Anderson	75.70
3	3	Wes	Anderson	209.90
4	4	Richard	Ayoade	0.90
5	5	Luc	Besson	264.40
6	6	James	Cameron	2187.30

22.4 Get Total Revenues for Each Director

```
WITH cte_movie_count AS
(
SELECT
d.director_id,
SUM(COALESCE(r.revenues_domestic,0) +
COALESCE(r.revenues_international,0)) AS total_revenues
FROM directors d
INNER JOIN movies mv ON mv.director_id = d.director_id
INNER JOIN movies_revenues r ON r.movie_id = mv.movie_id
GROUP BY d.director_id
)
```

```
SELECT
d.director_id,
d.first_name,
d.last_name,
cte.total_revenues
FROM cte_movie_count cte
INNER JOIN directors d ON d.director_id = cte.director_id;
```

I can then query from the results of the CTE and join it to any existing table to produce whatever columns I want.

Note how this time we see directors who's movies have not made any money from international or domestic sales.

This more refined CTE query is using coalesce to handle the Nulls

	director_id [PK] integer	total_revenues numeric
5	7	84.20
6	10	0
7	9	83.40
8	35	13.10
9	15	822.20

Using the colasece to convert null values to zero.

	director_id [PK] integer	first_name character varying (150)	last_name character varying (150)	total_revenues numeric
9	9	Francis	Ford Coppola	83.40
10	10	Kinji	Fukasaku	0
11	11	Florian	Henckel von Donnersmarck	77.40
12	12	Terry	Jones	19.60
13	13	Stanley	Kubrick	233.20
14	14	John	Lasseter	374.50
15	15	Ang	Lee	822.20
16	16	Bruce	Lee	0

22.5 Simultaneously DELETE INSERT via CTE

Lets take a scenario:

We have two tables:

- articles main articles
- Articles_delete all articles that are deleted

We want to delete some records from the articles table and we want all the records that we have deleted to be inserted into the 'articles_deleted' table.

We will use a CTE to reach this goal

```
INSERT INTO articles (title) VALUES  
('article1'),('article2'),('article3'),('article4');
```

```
SELECT * FROM articles;
```

	article_id [PK] integer	title character varying (100)
1	1	article1
2	2	article2
3	3	article3
4	4	article4

```
SELECT* FROM articles_delete;
```

	article_id integer	title character varying (100)

The aim is to delete rows from articles and insert that data into articles_deleted, i.e move rows from one table to another.

```
CREATE TABLE articles (  
    article_id SERIAL PRIMARY KEY,  
    title VARCHAR(100)  
);
```

```
CREATE TABLE articles_delete AS  
SELECT * FROM articles LIMIT 0;
```

This will copy the table structure of 'articles' for 'articles_delete' with the LIMIT 0 ensuring that no rows are copied in from 'articles'

Inserting some data and verifying it.

```
WITH cte_delete_articles AS
(
DELETE FROM articles
WHERE article_id = 1
RETURNING *
)
```

```
INSERT INTO articles_delete
SELECT * FROM cte_delete_articles;
```

The CTE contains a delete query to erase the row where article_id = 1. This query is RETURNING the result of this query.

Now I can insert into the 'articles_delete' table using by selecting the returned result of the CTE as the insert data.

```
SELECT * FROM articles;
```

	article_id [PK] integer	title character varying (100)
1	2	article2
2	3	article3
3	4	article4

```
SELECT* FROM articles_delete;
```

	article_id integer	title character varying (100)
1	1	article1

With the help of the CTE one row has 'moved' from articles to articles_delete.

```
CREATE TABLE articles_insert AS
SELECT * FROM articles LIMIT 0;
```

Aim is to move all articles in 'articles' table to 'articles_insert' table

```
WITH cte_move_articles AS
(
DELETE FROM articles
RETURNING *
)
INSERT INTO articles_insert
SELECT * FROM cte_move_articles;
```

```
SELECT * FROM articles;
```

	article_id [PK] integer	title character varying (100)

```
SELECT* FROM articles_insert;
```

	article_id integer	title character varying (100)
1	2	article2
2	3	article3
3	4	article4

22.6 Recursive CTEs

1. CTEs that calls itself until a condition is met is called recursive.
2. Can be used to work with hierarchical data
3. The traditional solution would be to involve some kind of iteration, probably by means of a cursor that iterates one tuple at a time over the whole result set.
4. The logic of recursive CTE is like a 'for loop' in programming languages.
5. When we use CTEs, it is **important to avoid infinite loops**. These can happen if the recursive does not end properly.

```
WITH RECURSIVE cte_name AS
(
cte_query_definition - non recursive term
UNION [ALL]
cte_query_definition - recursive term
exit conditions
)
SELECT * FROM cte_name;
```

22.7 Parent-Child relationship via CTEs

Create a number series from 10 to 50

```
WITH RECURSIVE series (list_num) AS
(
  -- non recursive statement
  SELECT 10
  UNION
  -- recursive statement
  -- 10 + 5 ... LOOP until <= 50
  SELECT list_num + 5 FROM series
  WHERE list_num + 5 <= 50 -- Exit condition
)
SELECT list_num FROM series;
```

Call the recursive CTE

The CTE contains two parts:

A non recursive statement to select 10.

A recursive statement to add 5 and very important is the exit condition so it does not run for infinity. In this case the exit condition is the where clause.

Both statements are joined by a union.

	list_num integer
1	10
2	15
3	20
4	25
5	30
6	35
7	40
8	45
9	50

Parent-Child relationship Example

```
CREATE TABLE items (
  pk SERIAL PRIMARY key,
  name TEXT NOT NULL,
  parent INT
)
```

```
SELECT * from items;
```

	pk [PK] integer	name text	parent integer
1	1	vegetables	0
2	2	fruits	0
3	3	apple	2
4	4	banana	2

```
INSERT INTO items (pk,name,parent) VALUES
(1,'vegetables',0),
(2,'fruits',0),
(3,'apple',2),
(4,'banana',2);
```

At the tree level all rows are equal but 'apple' and 'banana' are children of the parent 'fruit'. Lets show this recursively.

The desired output is a table showing two columns:

```
WITH RECURSIVE cte_tree AS
(
  -- non recursive statement
  -- All parent info
  SELECT
  name,
  pk,
  1 AS tree_level
  FROM items
  WHERE parent = 0
  UNION
  -- recursive statment
  -- loop to get all children of parent
  SELECT
  tt.name || '-> ' || ct.name,
  ct.pk,
  tt.tree_level + 1
  FROM items ct
  JOIN cte_tree tt ON tt.pk = ct.parent
)
SELECT
cte_tree.tree_level,
name
FROM cte_tree
ORDER BY cte_tree.tree_level;
```

Tree_level	name
1	vegetable
1	fruits
2	fruits -> apple
2	fruits -> banana

	tree_level integer	name text
1	1	vegetables
2	1	fruits
3	2	fruits-> apple
4	2	fruits-> banana

23. Grouping Sets

[23.1 Introduction to Summarization](#)

[23.2 Sub-totals on Group Sets](#)

[23.3 Adding sub-totals with ROLLUP](#)

23.4 Using Grouping with ROLLUP

23.1 Introduction to Summarization

1. When we aggregate data we normally remove the detail data that lies below the summarised data.
2. The whole point of aggregation is to replace the detailed data with summarisation. This is where the sub-totals comes in.
3. GROUP BY is used to group the data together. However when we summarise, we may need 'grouping sets'.
4. A group set is a set of columns by which you group
5. The PostgreSQL ROLLUP is a subclause of the GROUP BY clause that offers a shorthand for defining 'multiple grouping sets'.

23.2 Sub-totals on Group Sets

```
SELECT * FROM courses
ORDER BY course_level, sold_units;
```

```
CREATE TABLE courses (
  course_id SERIAL PRIMARY KEY,
  course_name VARCHAR(100) NOT NULL,
  course_level VARCHAR(100) NOT NULL,
  sold_units INT NOT NULL
);
```

	course_id [PK] integer	course_name character varying (100)	course_level character varying (100)	sold_units integer
1	3	Introduction to Python	Basic	200
2	2	Data Science Bootcamp	Premium	50
3	1	Machine learning with Pyth...	Premium	100
4	4	Understanding Mongo DB	Premium	100
5	5	Algorithm Design in Python	Premium	200

```
INSERT INTO courses (course_name,course_level,sold_units) VALUES
('Machine learning with Python', 'Premium', 100),
('Data Science Bootcamp', 'Premium', 50),
('Introduction to Python', 'Basic', 200),
('Understanding Mongo DB', 'Premium', 100),
('Algorithm Design in Python', 'Premium', 200);
```

This is a detail level,
shows all rows.

```
SELECT
course_level,
course_name,
SUM(sold_units) AS total_sold
FROM courses
GROUP BY course_level, course_name;
```

This is a detail level, still shows all rows.

	course_level character varying (100)	course_name character varying (100)	total_sold bigint
1	Premium	Understanding Mongo DB	100
2	Basic	Introduction to Python	200
3	Premium	Machine learning with Pyth...	100
4	Premium	Algorithm Design in Python	200
5	Premium	Data Science Bootcamp	50

```
SELECT
course_level,
SUM(sold_units) AS total_sold
FROM courses
GROUP BY course_level;
```

	course_level character varying (100)	total_sold bigint
1	Basic	200
2	Premium	450

If I remove course
name then I get a
summarisation view

```
SELECT
course_level,
course_name,
SUM(sold_units) AS total_sold
FROM courses
GROUP BY
ROLLUP (course_level, course_name);
```

The rollup is being performed on course level and course name

The rollup of the course name is the row **1** where it is summing the total of all the courses.

The rollup of course_level is rows **2** and **3** where 'basic' is summed then premium is summed.

```
SELECT
course_level,
course_name,
SUM(sold_units) AS total_sold
FROM courses
GROUP BY
course_level,
ROLLUP (course_name)
ORDER BY course_level, course_name;
```

The rollup is used to create totals for each course level.

	course_level character varying (100)	course_name character varying (100)	total_sold bigint
1	[null]	[null]	1 650
2	Premium	Understanding Mongo DB	100
3	Basic	Introduction to Python	200
4	Premium	Machine learning with Pyth...	100
5	Premium	Algorithm Design in Python	200
6	Premium	Data Science Bootcamp	50
7	Basic	[null]	2 200
8	Premium	[null]	3 450

	course_level character varying (100)	course_name character varying (100)	total_sold bigint
1	Basic	Introduction to Python	200
2	Basic	[null] total	200
3	Premium	Algorithm Design in Python	200
4	Premium	Data Science Bootcamp	50
5	Premium	Machine learning with Python	100
6	Premium	Understanding Mongo DB	100
7	Premium	[null] total	450

```
SELECT
course_level,
course_name,
SUM(sold_units) AS total_sold
FROM courses
GROUP BY
ROLLUP (course_level, course_name)
ORDER BY course_level, course_name;
```

	course_level character varying (100)	course_name character varying (100)	total_sold bigint
1	Basic	Introduction to Python	200
2	Basic	[null] total	200
3	Premium	Algorithm Design in Python	200
4	Premium	Data Science Bootcamp	50
5	Premium	Machine learning with Python	100
6	Premium	Understanding Mongo DB	100
7	Premium	[null] total	450
8	[null]	[null] Grand total	650

23.3 Adding sub-totals with ROLLUP

```
SELECT * FROM inventory;
```

```
CREATE TABLE inventory (
inventory_id SERIAL PRIMARY KEY,
category VARCHAR(100) NOT NULL,
sub_category VARCHAR(100) NOT NULL,
product VARCHAR(100) NOT NULL,
quantity INT
);
```

	inventory_id [PK] integer	category character varying (100)	sub_category character varying (100)	product character varying (100)	quantity integer
1	1	Furniture	Chair	Black	10
2	2	Furniture	Chair	Brown	10
3	3	Furniture	Desk	Blue	10
4	4	Equipment	Computer	Mac	5
5	5	Equipment	Computer	PC	5
6	6	Equipment	Computer	Dell	10

```
INSERT INTO inventory (category, sub_category, product, quantity) VALUES
('Furniture', 'Chair', 'Black', 10),
('Furniture', 'Chair', 'Brown', 10),
('Furniture', 'Desk', 'Blue', 10),
('Equipment', 'Computer', 'Mac', 5),
('Equipment', 'Computer', 'PC', 5),
('Equipment', 'Monitor', 'Dell', 10);
```

```
SELECT
category,
sub_category,
SUM(quantity) AS "Quantity"
FROM inventory
GROUP BY
category,
sub_category
ORDER BY
category,
sub_category;
```

	catagory character varying (100)	sub_catagory character varying (100)	Quantity bigint
1	Equipment	Computer	10
2	Equipment	Monitor	10
3	Furniture	Chair	20
4	Furniture	Desk	10

Grouping by
'category' and
'sub_category'

```
SELECT
category,
SUM(quantity) AS "Quantity"
FROM inventory
GROUP BY
category;
```

	catagory character varying (100)	Quantity bigint
1	Furniture	30
2	Equipment	20

Grouping by 'category'

```
SELECT
category,
sub_category,
SUM(quantity) AS "Quantity"
FROM inventory
GROUP BY
ROLLUP (catagory, sub_catagory)
ORDER BY
category,
sub_catagory;
```

	catagory character varying (100)	sub_catagory character varying (100)	Quantity bigint
1	Equipment	Computer	10
2	Equipment	Monitor	10
3	Equipment	[null]	20
4	Furniture	Chair	20
5	Furniture	Desk	10
6	Furniture	[null]	30
7	[null]	[null]	50

Grouping
by sub
totals and
grand total

23.4 Using Grouping with ROLLUP

Can we improve the aesthetic of this table so that the NULL values are more human readable?

We will use GROUPING function with ROLLUP

GROUPING is a special aggregate function that works in conjunction with ROLLUP function

GROUPING returns 0 or 1. The function returns 0 if it is not a subtotal-row for given column_name

```
SELECT
category,
sub_category,
SUM(quantity) AS "Quantity",
GROUPING(category) AS
"Category Grouping",
GROUPING(sub_category) AS
"Sub-Category grouping"
FROM inventory
GROUP BY
ROLLUP (category,
sub_category)
ORDER BY
category,
sub_category;
```

	category character varying (100)	sub_category character varying (100)	Quantity bigint	Category Grouping integer	Sub-Category grouping integer
1	Equipment	Computer	10	0	0
2	Equipment	Monitor	10	0	0
3	Equipment	[null]	20	0	1 1
4	Furniture	Chair	20	0	0
5	Furniture	Desk	10	0	0
6	Furniture	[null]	30	0	2 1
7	[null]	[null]	50	3 1	1

I have adding additional GROUPING() columns which contain a 1 if they are sub-totals and a 0 if they are not.

```

SELECT
COALESCE(catagory, '') AS "Catagory"
COALESCE(sub_catagory, '') AS "Sub-Catagory"
SUM(quantity) AS "Quantity",
CASE
WHEN GROUPING(catagory) = 1 THEN "grand-Total" ELSE ''
END AS "Sub-Total/Total"
FROM inventory
GROUP BY
ROLLUP (catagory, sub_catagory)
ORDER BY
catagory,
sub_catagory;

```

The case statement is used to create a new column called 'Sub-Total/Total' where a phrase 'grand total' is output of the ROLLUP

	Catagory character varying	Sub-Catagory character varying	Quantity bigint	Sub-Total/Total text
1	Equipment	Computer	10	
2	Equipment	Monitor	10	
3	Equipment		20	
4	Furniture	Chair	20	
5	Furniture	Desk	10	
6	Furniture		30	
7			50	Grand-Total

I am using COALESCE() to output the value of category or nothing if it is NULL. This removes the ugly Nulls from the table

Further refinement of the case statement, coalesce and grouping produces a human readable table.

```

SELECT
CASE
WHEN GROUPING(catagory) = 1 THEN 'grand-Total ==>'
WHEN GROUPING(sub_catagory) = 1 THEN 'Sub-Total ==>'
ELSE COALESCE(catagory, '')
END AS "Catagory",
COALESCE(sub_catagory, '') AS "Sub-Catagory",
SUM(quantity) AS "Quantity"
FROM inventory
GROUP BY
ROLLUP (catagory, sub_catagory)
ORDER BY
catagory,
sub_catagory;

```

	Catagory character varying	Sub-Catagory character varying	Quantity bigint
1	Equipment	Computer	10
2	Equipment	Monitor	10
3	Sub-Total ==>		20
4	Furniture	Chair	20
5	Furniture	Desk	10
6	Sub-Total ==>		30
7	grand-Total ==>		50

24. Window Functions

[24.1 Introduction to Window Functions](#)

[24.2 Analyse Global Trade Data](#)

[24.3 Using Aggregate Functions](#)

[24.4 Group by Rollup](#)

[24.5 Rollup . Grouping Multiple Columns](#)

[24.6 Group By Cube](#)

[24.7 Group by Grouping Sets](#)

[24.8 Query Performance Check](#)

[24.9 Using Filter Clause](#)

[24.10 Using Window Function](#)

[24.11 Partitioning the Data](#)

[24.12 Set Data into Millions Format](#)

[24.13 Ordering Inside Window](#)

[24.14 Sliding Dynamic Window](#)

[24.15 Understanding Window Frames](#)

[24.16 Rows & Range Indicators](#)

[24.17 Using Window](#)

[24.18 Using Window with Rows Between](#)

[24.19 Rank & Dense Rank functions](#)

[24.20 Ntile Function](#)

[24.21 Lead & Lag Functions](#)

[24.22 First Value, Last Value & Nth Value](#)

[24.23 Row Number Function](#)

[24.24 Finding Correlations](#)

[24.25 Row Number\(\) – Window with Partition Datasets](#)

24. Window Functions

[24.26 Row Number\(\) – Reverse Fields with Order By](#)

[24.27 Row Number\(\) – Find Nth Record](#)

[24.28 Row Number\(\) – Find Distinct with Subquery](#)

[24.29 Row Number\(\) – Pagination Technique](#)

[24.30 Using over Function to Calculate Percentage](#)

[24.31 Calculate Difference Compared to Average](#)

[24.32 Calculating Cumulative Totals](#)

[24.33 Using Lead to Compare with Next Values](#)

[24.34 Comparing with Least number](#)

[24.35 Window Function Summary](#)

[24.36 Planning Tips for Using Windows Functions](#)

[24.37 Difference between Rank & Dense Rank](#)

[24.38 Getting Rank to compute group and Global Ranks](#)

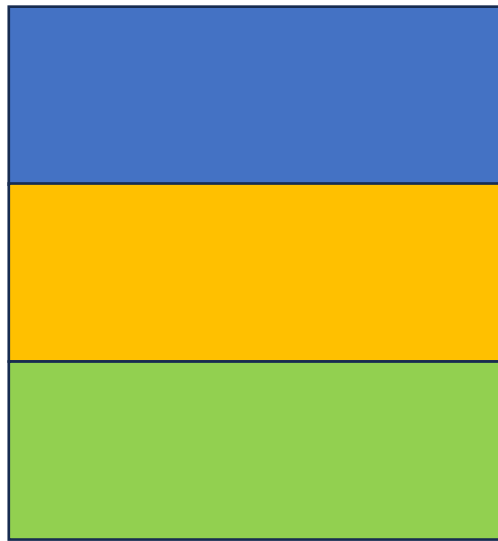
[24.39 Using Partition By for Group Averages](#)

[24.40 Using With Clause to Create Own Data](#)

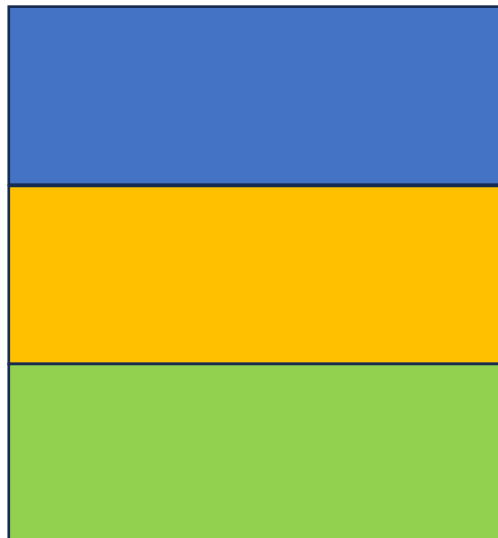
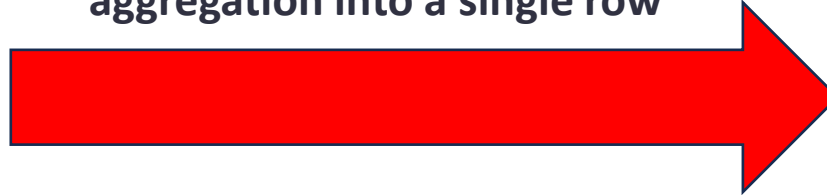
[24.41 Using With Clause to set Order By Values](#)

[24.42 Delete and Insert in One query using With](#)

24.1 Introduction to Window Functions



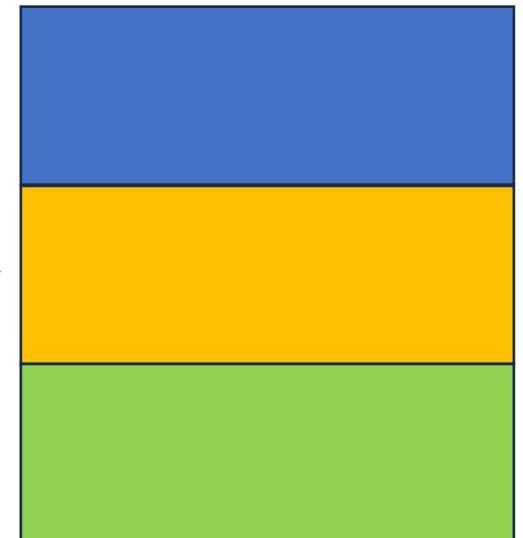
Data is flattened through
aggregation into a single row



Create Aggregation without
Flattening Data into Single Rows

USING WINDOWS FUNCTIONS

Windows functions don't
flatten data but put it in
different windows



24.2 Analyse Global Trade Data

The sample data is a 4000+ row open source dataset taken from <https://comtradeplus.un.org/TradeFlow>



trades.sql

```
CREATE TABLE trades (  
  region TEXT,  
  country TEXT,  
  Year INT,  
  imports NUMERIC(50,0),  
  exports NUMERIC(50,0)  
);
```

```
\i C:/Users/ellio/Documents/CODING-LESSONS/15-  
Advanced_PostgreSQL/resources/trades.sql  
INSERT 0 1  
Query returned successfully in 191 msec.
```

Confirm that all 4000+ rows have been inserted into the table.

The data has been massaged to fit the lessons in this chapter and may not reflect the original source data from comtradeplus.un.org

```
SELECT * FROM trades;
```

Data Output Messages Notifications

	region text	country text	year integer	imports numeric (50)	exports numeric (50)
1	ASIA	Afghanistan	2008	3019860129	540065594
2	ASIA	Afghanistan	2009	3336434781	403441006
3	ASIA	Afghanistan	2010	5154249867	388483635
4	ASIA	Afghanistan	2011	6390310947	375850935
5	ASIA	Afghanistan	2012	6204984101	428902710

Understand the data in the trades table

```
SELECT DISTINCT (region)
FROM trades
ORDER BY region;
```

	region text
1	AFRICA
2	ASIA
3	CARIBBEAN
4	CENTRAL AMERICA
5	EUROPE
6	MIDDLE EAST
7	NORTH AMERICA
8	OCEANIA
9	SOUTH AMERICA

```
SELECT DISTINCT (country)
FROM trades
ORDER BY country;
```

	country text
1	Afghanistan
2	Albania
3	Algeria
4	Andorra
5	Angola
6	Anguilla
7	Antigua and Barbuda
8	Argentina
9	Armenia

```
SELECT
MIN(year),
MAX(year)
FROM trades;
```

	min integer	max integer
1	1988	2019

The table contains imports and exports for 206 countries in 9 geographic regions of the world from the year 1988 to 2019

```
SELECT * FROM trades WHERE country = 'USA' ORDER BY year;
```

	region text	country text	year integer	imports numeric (50)	exports numeric (50)
1	SOUTH AMERICA	USA	1991	508944033884	421555416008
2	SOUTH AMERICA	USA	1992	553496473571	447330086418
3	SOUTH AMERICA	USA	1993	603153591727	464757163716
4	SOUTH AMERICA	USA	1994	689029922860	512336854960
5	SOUTH AMERICA	USA	1995	770821455872	582964674560

Each row represents economic data about exports and imports for a country for a given year.

24.3 Using Aggregate Functions

```
SELECT
MIN(imports) / 100000 AS "Min-Imports ($mil)",
MAX(imports) / 100000 AS "Max-Imports ($mil)",
AVG(imports) / 100000 AS "Avg-Imports ($mil)"
FROM trades;
```

Min. max and avg imports for the whole world

	Min-Imports (\$mil) numeric	Max-Imports (\$mil) numeric	Avg-Imports (\$mil) numeric
1	0.000000000000000000000000	26114324.9015700000000	733252.900661650485

```
SELECT
region,
Round(MIN(imports) / 100000,2) AS "Min-Imports ($mil)",
Round(MAX(imports) / 100000,2) AS "Max-Imports ($mil)",
Round(AVG(imports) / 100000,2) AS "Avg-Imports ($mil)"
FROM trades
GROUP BY region
ORDER BY 3 DESC;
```

Min. max and avg imports grouped by region.

	region text	Min-Imports (\$mil) numeric	Max-Imports (\$mil) numeric	Avg-Imports (\$mil) numeric
1	SOUTH AMERICA	0.00	26114324.90	1529689.52
2	ASIA	0.00	21349826.15	1154762.97
3	EUROPE	0.00	12927260.52	1136068.35
4	CENTRAL AMERICA	2555.53	4642942.63	388539.80
5	NORTH AMERICA	0.00	4630889.77	709934.13
6	MIDDLE EAST	0.00	2986087.29	560470.57
7	AFRICA	0.00	1041804.11	68873.77
8	OCEANIA	325.77	468778.56	56869.54
9	CARIBBEAN	0.00	221810.80	18500.95

By using some simple aggregate functions we can draw some basic conclusions such as South America is the region with the highest imports.

24.4 Group by Rollup

```
SELECT
region,
Round(AVG(imports)/1000000000,2) AS "avg
($bil)"
FROM trades
GROUP BY ROLLUP (region)
ORDER BY 1,2 DESC;
```

The average for global imports across all regions is 73.33 billion dollars.

	region text	avg (\$bil) numeric
1	AFRICA	6.89
2	ASIA	115.48
3	CARIBBEAN	1.85
4	CENTRAL AMERICA	38.85
5	EUROPE	113.61
6	MIDDLE EAST	56.05
7	NORTH AMERICA	70.99
8	OCEANIA	5.69
9	SOUTH AMERICA	152.97
10	[null]	73.33

24.5 Rollup . Grouping Multiple Columns

```
SELECT
region,
country,
Round(AVG(imports)/1000000000,2) AS "avg ($bil)"
FROM trades
WHERE country IN('USA','Argentina','Singapore','Brazil')
GROUP BY ROLLUP (region, country)
ORDER BY 1;
```

The rollup columns with null are the average imports for each region containing the listed countries.

	region text	country text	avg (\$bil) numeric
1	ASIA	Singapore	209.19
2	ASIA	[null]	209.19
3	SOUTH AMERICA	Argentina	40.76
4	SOUTH AMERICA	Brazil	103.65
5	SOUTH AMERICA	USA	1590.28
6	SOUTH AMERICA	[null]	579.68
7	[null]	[null]	482.35

24.6 Group By Cube

The CUBE allows you to generate multiple grouping sets

GROUP BY CUBE (col1, col2, ...)

The query generates all possible grouping sets based on the dimension columns specified in the CUBE.

IF the number of columns in the cube is n then we will generate 2 to the power of n combinations.

```
SELECT
region,
country,
Round(AVG(imports)/1000000000,2) AS "avg ($bil)"
FROM trades
WHERE country
IN('USA','France','Germany','Brazil')
GROUP BY CUBE (region, country);
```

The average for global imports across all countries is 719.17 billion dollars. The cube function is also performing an average on each country and region of countries

GROUP BY CUBE (c1, c2, c3)

(c1, c2, c3)

(c1, c2)

(c1, c3)

(c2, c3)

(c1)

(c2)

(c3)

() – grand total

	region text	country text	avg (\$bil) numeric
1	[null]	[null]	719.17
2	EUROPE	France	481.42
3	SOUTH AMERICA	Brazil	103.65
4	SOUTH AMERICA	USA	1590.28
5	SOUTH AMERICA	[null]	822.19
6	EUROPE	[null]	481.42
7	[null]	USA	1590.28
8	[null]	France	481.42
9	[null]	Brazil	103.65

24.7 Group by Grouping Sets

```
SELECT
region,
country,
Round(AVG(imports)/1000000000,2) AS "avg ($bil)"
FROM trades
WHERE country
IN('USA','France','Germany','Brazil')
GROUP BY
GROUPING SETS (
(),
country,
region
);
```

	region text	country text	avg (\$bil) numeric
1	[null]	[null]	739.71
2	[null]	USA	1590.28
3	[null]	Germany	800.65
4	[null]	France	481.42
5	[null]	Brazil	103.65
6	SOUTH AMERICA	[null]	822.19
7	EUROPE	[null]	649.74

24.8 Query Performance Check

```
EXPLAIN SELECT
region,
country,
Round(AVG(imports)/1000000000,2) AS "avg ($bil)"
FROM trades
WHERE country
IN('USA','France','Germany','Brazil')
GROUP BY
GROUPING SETS (
(),
country,
region
);
```

Are these expensive queries to execute?

	QUERY PLAN text
1	MixedAggregate (cost=0.00..107.06 rows=99 width=49)
2	Hash Key: country
3	Hash Key: region
4	Group Key: ()
5	-> Seq Scan on trades (cost=0.00..103.89 rows=115 wid...
6	Filter: (country = ANY ('{USA,France,Germany,Brazil}':....

What I can see is that PostgreSQL is using a mixed aggregate with hash key on country and region. It is also using a sequence scan on the group. But compared to achieving the same results with a complex union query it is cheaper.


```
SHOW enable_hashagg;
```

	enable_hashagg
	text
1	on

```
SET enable_hashagg TO off;
```

Now the explain analyze of the same query is showing that PostgreSQL is using the group aggregate which has:
Planning time: 0.179ms
Execution time: 0.907ms

```
SET enable_hashagg TO on;
```

```
SHOW enable_hashagg;
```

	enable_hashagg
	text
1	on

Now the explain analyze of the same query is showing that PostgreSQL is using the group mixed aggregate with slightly faster times.
Planning time: 0.106ms
Execution time: 0.916ms

	QUERY PLAN
	text
1	GroupAggregate (cost=107.83..115.22 rows=99 width=49) (actual time=0.823..0.877 rows=7 loops=1)
2	Group Key: country
3	Group Key: ()
4	Sort Key: region
5	Group Key: region
6	-> Sort (cost=107.83..108.11 rows=115 width=25) (actual time=0.785..0.791 rows=115 loops=1)
7	Sort Key: country
8	Sort Method: quicksort Memory: 30kB
9	-> Seq Scan on trades (cost=0.00..103.89 rows=115 width=25) (actual time=0.115..0.755 rows=115 l...
10	Filter: (country = ANY ('{USA,France,Germany,Brazil}':text[]))
11	Rows Removed by Filter: 4211
12	Planning Time: 0.179 ms
13	Execution Time: 0.907 ms

	QUERY PLAN
	text
1	MixedAggregate (cost=0.00..107.06 rows=99 width=49) (actual time=0.872..0.897 rows=7 loops=1)
2	Hash Key: country
3	Hash Key: region
4	Group Key: ()
5	Batches: 1 Memory Usage: 40kB
6	-> Seq Scan on trades (cost=0.00..103.89 rows=115 width=25) (actual time=0.114..0.778 rows=115 loop...
7	Filter: (country = ANY ('{USA,France,Germany,Brazil}':text[]))
8	Rows Removed by Filter: 4211
9	Planning Time: 0.106 ms
10	Execution Time: 0.916 ms

When doing grouping queries on large datasets it is worth doing an explain analyze to see which type of query is fastest and has lowest cost.

24.9 Using Filter Clause

The FILTER clause allows you to do a 'selective aggregate'.

It aggregate the group sets based on a filter condition.

Allows you to selectively pass data to those aggregates.

Aggregate_function
FILTER (WHERE condition)

```
SELECT
region,
AVG(imports) AS "avg All",
AVG(exports) FILTER (WHERE year < 1995) AS "Avg old",
AVG(exports) FILTER (WHERE year >= 1995) AS "Avg latest"
FROM trades
GROUP BY ROLLUP (region);
```

Get average exports per
each region for all periods,
for export year < 1995,
year >= 1995 as latest.

	region text	avg All numeric	Avg old numeric	Avg latest numeric
1	[null]	73325290066.16504854	42905138774.45808383	74914795899.38702405
2	NORTH AMERICA	70993412846.58461538	62932794640.69230769	75693135884.41880342
3	SOUTH AMERICA	152968951703.49295775	49195642528.37777778	118069071091.03548387
4	ASIA	115476296703.28694581	59879433497.46250000	129413458239.61338798
5	CENTRAL AMERICA	38853979545.29629630	12904661532.05555556	36966773489.08080808
6	OCEANIA	5686953677.41269841	7584794664.12500000	4799598057.74585635
7	EUROPE	113606835430.76657825	57235575559.22549020	121237992372.68610301
8	AFRICA	6887377415.18686869	3583919301.04761905	6905984827.79852321
9	MIDDLE EAST	56047057495.02671756	15037068334.60000000	77776616795.81376518
10	CARIBBEAN	1850095330.37759336	87256288.3636363636	747671070.23913043

24.10 Using Window Function

Aggregate functions Take many rows and turn them into fewer aggregated rows flattening the data

A windowing Function It compares the current row with all rows in the group

```
SELECT AVG(exports) FROM trades;
```

	avg numeric
1	72443407670.13915858

72443407670.13915858

Lets partition the data with OVER clause –

Aggregate_function OVER (PARTITION BY group_name)

```
SELECT country, year, imports, exports, AVG(exports) OVER() AS avg_exports FROM trades;
```

	country text	year integer	imports numeric (50)	exports numeric (50)	avg_exports numeric
1	Afghanistan	2008	3019860129	540065594	72443407670.13915858
2	Afghanistan	2009	3336434781	403441006	72443407670.13915858
3	Afghanistan	2010	5154249867	388483635	72443407670.13915858
4	Afghanistan	2011	6300310047	375850035	72443407670.13915858

With one SQL command an aggregate column can be added and it can be compared with the rest of the data. I.e. country exports per year can be compared with average exports of the whole table.

24.11 Partitioning the Data

```
SELECT country, year, imports, exports,  
AVG(exports) OVER(PARTITION BY country) AS avg_exports  
FROM trades;
```

	country text	year integer	imports numeric (50)	exports numeric (50)	avg_exports numeric
9	Afghanistan	2008	3019860129	540065594	527461567.00000000
10	Afghanistan	2011	6390310947	375850935	527461567.00000000
11	Albania	2005	2614347012	658233171	1179251626.43478261
12	Albania	2006	3057408426	792629671	1179251626.43478261
13	Albania	2007	4200864046	1077690359	1179251626.43478261

The OVER (PARTITION country) is calculating the average of the total for each country. i.e. is grouping the rows by country then calculating the average for each group.

```
SELECT country, year, imports, exports,  
AVG(exports) OVER(PARTITION BY year < 2000) AS avg_exports  
FROM trades;
```

	country text	year integer	imports numeric (50)	exports numeric (50)	avg_exports numeric
1	Zimbabwe	2019	4787446790	4279383537	81990553634.91364318
2	Afghanistan	2009	3336434781	403441006	81990553634.91364318
3	Afghanistan	2010	5154249867	388483635	81990553634.91364318
4	Afghanistan	2011	6390310947	375850935	81990553634.91364318
5	Afghanistan	2012	6204984101	428902710	81990553634.91364318

The OVER (PARTITION BY year < 2000) is calculating the average of the total for for all rows where the year is less than 2000.

24.12 Set Data into Millions Format

```
SELECT
imports,
round(imports/1000000,2)
FROM trades;
```

	imports numeric (50)	round numeric
1	3019860129	3019.86
2	3336434781	3336.43
3	5154249867	5154.25
4	6390310947	6390.31

Our data values when aggregating (MAX) are very large so we can divide values by a million and round.

```
UPDATE trades
SET
imports = ROUND(imports/1000000,2),
exports = ROUND(exports/1000000,2);
UPDATE 4326
Query returned successfully in 34 msec.
```

```
SELECT * FROM trades;
```

	region text	country text	year integer	imports numeric (50)	exports numeric (50)
1	AFRICA	Algeria	2008	39475	0
2	AFRICA	Algeria	2009	39258	0
3	AFRICA	Algeria	2010	41000	0
4	AFRICA	Algeria	2011	47220	0
5	AFRICA	Algeria	2012	50369	0

Above I have converted all rows of imports and exports into millions

24.13 Ordering Inside Window

Aggregate_function OVER (PARTITION BY
group_name ORDER BY column)

```
SELECT
country,
year,
exports,
MIN(exports) OVER (PARTITION BY country ORDER BY year)
FROM trades
WHERE year > 2001
AND country IN ('USA', 'France');
```

```
SELECT
country,
year,
exports,
MAX(exports) OVER (PARTITION BY country ORDER BY year)
FROM trades
WHERE
year > 2001
AND country IN ('USA', 'France');
```

The MAX is calculating from the start of the partition so on row 19 which is the start of the USA partition the MAX is derived from one row. On Row 20, the second year for USA the Max is calculated on row 19 and 20. Therefore as it works down the partition the MAX value will change with each row.

	country text	year integer	exports numeric (50)	min numeric
1	France	2002	304892	304892
2	France	2003	358132	304892
3	France	2004	413708	304892
4	France	2005	434354	304892
5	France	2006	479013	304892
6	France	2007	539731	304892
7	France	2008	594505	304892
8	France	2009	664112	304892

	country text	year integer	exports numeric (50)	max numeric
18	France	2019	556364	594505
19	USA	2002	693068	693068
20	USA	2003	724737	724737
21	USA	2004	814844	814844
22	USA	2005	901041	901041
23	USA	2006	1037029	1037029
24	USA	2007	1162538	1162538
25	USA	2008	1299899	1299899
26	USA	2009	1356710	1356710

24.14 Sliding Dynamic Window

```
SELECT
country,
year,
exports,
MIN(exports) OVER (PARTITION BY country
                    ORDER BY year
                    ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING
                    ) AS avg_moving
FROM trades
FROM trades
WHERE year BETWEEN 2001 AND 2010
AND country IN('USA','France');
```

Rolling minimum

The moving minimum is calculated from the current row, previous row and following row.

	country text	year integer	exports numeric (50)	avg_moving numeric
1	France	2001	289599	289599
2	France	2002	304892	289599
3	France	2003	358132	304892
4	France	2004	413708	358132
5	France	2005	434354	413708
6	France	2006	479013	434354
7	France	2007	539731	479013
8	France	2008	594505	464113
9	France	2009	464113	464113

This minimum is calculated from the current row of 289599, the previous row of 0 and the following row of 304892

This minimum is calculated from the current row of 304892, the previous row of 289599 and the following row of 358132

This minimum is calculated from the current row of 289599, the previous row of 304892 and the following row of 413708


```

SELECT
country,
year,
exports,
AVG(exports) OVER (PARTITION BY country
                    ORDER BY year
                    ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING
                    ) AS avg_moving

FROM trades
FROM trades
WHERE year BETWEEN 2001 AND 2010
AND country IN('USA','France');

```

Rolling Average

The moving average is calculated from the current row, previous row and following row.

	country 	year 	exports  numeric (50)	avg_moving  numeric
1	France	2001	289599	297245.5000000000000
2	France	2002	304892	317541.0000000000000
3	France	2003	358132	358910.6666666666667
4	France	2004	413708	402064.6666666666667
5	France	2005	434354	442358.3333333333333
6	France	2006	479013	484366.0000000000000
7	France	2007	539731	537749.6666666666667
8	France	2008	594505	532783.0000000000000
9	France	2009	464113	523423.0000000000000

This average is calculated from the current row of 289599, the previous row of 0 and the following row of 304892

This average is calculated from the current row of 304892, the previous row of 289599 and the following row of 358132

This average is calculated from the current row of 289599, the previous row of 304892 and the following row of 413708


```

SELECT
country,
year,
exports,
AVG(exports) OVER (PARTITION BY country
                    ORDER BY year
                    ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING
                    EXCLUDE CURRENT ROW
                    ) AS avg_moving
FROM trades
FROM trades
WHERE year BETWEEN 2001 AND 2010
AND country IN('USA','France');

```

Rolling Average exclude current row

The moving average is calculated from the previous row and following row.

	country text	year integer	exports numeric (50)	avg_moving numeric
1	France	2001	289599	304892.000000000000
2	France	2002	304892	323865.500000000000
3	France	2003	358132	359300.000000000000
4	France	2004	413708	396243.000000000000
5	France	2005	434354	446360.500000000000
6	France	2006	479013	487042.500000000000
7	France	2007	539731	536759.000000000000
8	France	2008	594505	501922.000000000000

This average is calculated from the previous row of 0 and the following row of 304892

This average is calculated from the previous row of 289599 and the following row of 358132

This average is calculated from the previous row of 304892 and the following row of 413708

24.15 Understanding Window Frames

Window frames are used to indicate how many rows around the current row, the window function should include.

Specify window frame via:

ROWS or RANGE indicators

BETWEEN start of the frame and the end of the frame

Window frames in window functions use UNBOUND PRECEDING by default

RANGE BETWEEN UNBOUND PRECEDING AND CURRENT ROW

Possible Frame combinations:

UNBOUND Everything

UNBOUNDED 1 PROCEEDING
PROCEEDING

UNBOUNDED 1 FOLLOWING
0 FOLLOWING

CURRENT ROW

ROWS		
1		
2		
3	ROWS BETWEEN UNBOUND PRECEDING	Everything before the current row.
4		
5		Upto thre rows will be in use: Current row, row before, row after
6	ROWS BETWEEN 1 PROCEEDING AND 1 FOLLOWING	
7		
8		
9	ROWS BETWEEN UNBOUNDED FOLLOWING	Everything after the current row.
10		
11		

I am generating a series of numbers between 1 and 3 called column x.

```
SELECT
*,
ARRAY_AGG(x) OVER (ORDER BY x)
FROM generate_series(1,3) AS x;
```

	x integer	array_agg integer[]
1	1	{1}
2	2	{1,2}
3	3	{1,2,3}

I am then aggregating the array with over so as each row passes the array picks up the current x and all preceding x's because unbound preceding is the default.

Here I am aggregating the array with current row and all following rows.

```
SELECT
*,
ARRAY_AGG(x) OVER (ORDER BY x
                    ROWS BETWEEN CURRENT ROW
                    AND UNBOUNDED FOLLOWING)
FROM generate_series(1,3) AS x;
```

	x integer	array_agg integer[]
1	1	{1,2,3}
2	2	{2,3}
3	3	{3}

Here I am aggregating the array with the current and proceeding rows and none following

```
SELECT
*,
ARRAY_AGG(x) OVER (ORDER BY x
                    ROWS BETWEEN UNBOUNDED PRECEDING
                    AND 0 FOLLOWING)
FROM generate_series(1,3) AS x;
```

	x integer	array_agg integer[]
1	1	{1}
2	2	{1,2}
3	3	{1,2,3}

Here I am aggregating the array with everything before and 2 proceeding

```
SELECT
*,
ARRAY_AGG(x) OVER (ORDER BY x
                    ROWS BETWEEN UNBOUNDED PRECEDING
                    AND 2 FOLLOWING)
FROM generate_series(1,5) AS x;
```

	x integer	array_agg integer[]
1	1	{1,2,3}
2	2	{1,2,3,4}
3	3	{1,2,3,4,5}
4	4	{1,2,3,4,5}
5	5	{1,2,3,4,5}

I am generating a series of numbers between 1 and 3.

```
SELECT  
ARRAY_AGG(x) OVER () AS frame  
FROM generate_series(1,3) AS t(x);
```

	frame integer[] 🔒
1	{1,2,3}
2	{1,2,3}
3	{1,2,3}

I am then aggregating the array as a frame where the frame covers the whole set of rows.

```
SELECT  
ARRAY_AGG(x) OVER () AS FRAME,  
SUM(x) OVER () AS sum  
FROM generate_series(1,3) AS t(x);
```

	frame integer[] 🔒	sum bigint 🔒
1	{1,2,3}	6
2	{1,2,3}	6
3	{1,2,3}	6

The second column is the sum of all rows in the frame

```
SELECT  
ARRAY_AGG(x) OVER () AS FRAME,  
SUM(x) OVER () AS sum,  
x::FLOAT/SUM(x) OVER () AS part_perc  
FROM generate_series(1,3) AS t(x);
```

	frame integer[] 🔒	sum bigint 🔒	part_perc double precision 🔒
1	{1,2,3}	6	0.16666666666666666
2	{1,2,3}	6	0.3333333333333333
3	{1,2,3}	6	0.5

The third column is the current row value of x divided by the sum of all rows.

$1 / 6 = 0.16666666666666666$
 $2 / 6 = 0.3333333333333333$
 $3 / 6 = 0.5$

The beauty and power of window frames is to be able to do all of this in one SQL command and to understand frames, ranges, proceeding and following.

24.16 Rows & Range Indicators

RANGE can only be used with UNBOUND

ROWS can actually be used for all of the options

How aggregations are treated differently

If the fields you use for ORDER BY do not contain a unique values for each row, then RANGE will combine all the rows it comes across for non-unique values rather than processing them one at a time.

ROWS will include all of the rows in the non-unique bunch but process each of them separately.

I am generating a series of numbers between 1 and 10 called X column.

```
SELECT
*,
x / 3 AS y,
ARRAY_AGG(x) OVER (ORDER BY x
                    ROWS BETWEEN 1 PRECEDING
                           AND 1 FOLLOWING) AS rows1
FROM generate_series(1,10) AS x;
```

The y column is populated with x divided by 3, which by default will round down to the nearest integer because float data type has not been specified.

Rows1 column contains the array of the x value of 1 row proceeding, the current row value of x and the following row value of x.

	x integer	y integer	rows1 integer[]
1	1	0	{1,2}
2	2	0	{1,2,3}
3	3	1	{2,3,4}
4	4	1	{3,4,5}
5	5	1	{4,5,6}
6	6	2	{5,6,7}
7	7	2	{6,7,8}
8	8	2	{7,8,9}
9	9	3	{8,9,10}
10	10	3	{9,10}

```

SELECT
*,
x / 3 AS y,
ARRAY_AGG(x) OVER (ORDER BY x
                    ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) AS rows1,
ARRAY_AGG(x) OVER (ORDER BY x
                    RANGE BETWEEN 1 PRECEDING AND 1 FOLLOWING) AS range1,
ARRAY_AGG(x/3) OVER (ORDER BY (x/3)
                     ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) AS rows2,
ARRAY_AGG(x/3) OVER (ORDER BY (x/3)
                     RANGE BETWEEN 1 PRECEDING AND 1 FOLLOWING) AS range2
FROM generate_series(1,10) AS x;

```

The values in column X are unique whereas the values in column Y (x/3) are not unique

Rows1 is taking current, one preceding and one following of x.

Range1 is taking current, one preceding and one following of x.

Rows2 is taking current, one preceding and one following of y.

For example row 1 of the table is taking y=0 and y=0. row 2 is taking y=0, y=0 and y=1.

Range2: because Y is not unique, RANGE will combine all the rows it comes across for non-unique values rather than processing them one at a time.

	x integer	y integer	rows1 integer[]	range1 integer[]	rows2 integer[]	range2 integer[]	🔒
1	1	0	{1,2}	{1,2}	{0,0}	{0,0,1,1,1}	
2	2	0	{1,2,3}	{1,2,3}	{0,0,1}	{0,0,1,1,1}	
3	3	1	{2,3,4}	{2,3,4}	{0,1,1}	{0,0,1,1,1,2,2,2}	
4	4	1	{3,4,5}	{3,4,5}	{1,1,1}	{0,0,1,1,1,2,2,2}	
5	5	1	{4,5,6}	{4,5,6}	{1,1,2}	{0,0,1,1,1,2,2,2}	
6	6	2	{5,6,7}	{5,6,7}	{1,2,2}	{1,1,1,2,2,2,3,3}	
7	7	2	{6,7,8}	{6,7,8}	{2,2,2}	{1,1,1,2,2,2,3,3}	
8	8	2	{7,8,9}	{7,8,9}	{2,2,3}	{1,1,1,2,2,2,3,3}	
9	9	3	{8,9,10}	{8,9,10}	{2,3,3}	{2,2,2,3,3}	
10	10	3	{9,10}	{9,10}	{3,3}	{2,2,2,3,3}	

24.17 Using Window

WINDOW function allows us to add columns to the result set that has been calculated on the fly.

Allows to pre-define a result set and use it anywhere (even multiple places) in every query.

Using multiple windows in a query

```
SELECT
wf1() OVER (PARTITION BY c1 ORDER BY c2),
wf2() OVER (PARTITION BY c1 ORDER BY c2)
FROM table_name;
```

```
SELECT
wf1() OVER w,
wf2() OVER w
FROM table_name
WINDOW w AS (PARTITION BY c1 ORDER BY c2);
```

Shortened format
of the query

```
SELECT
wf1() OVER w
FROM table_name
WINDOW w AS (PARTITION BY c1 ORDER BY c2);
```

Only one window
defined after

Example: Get min and max exports
per year per each country from year
2000 onwards in a single query

```
SELECT
country,
year,
exports,
MIN(exports) OVER (w),
MAX(exports) OVER (w)
FROM trades
WHERE country = 'USA'
WINDOW w AS (ORDER BY year);
```

	country text	year integer	exports numeric (50)	min numeric	max numeric
1	USA	1991	421555	421555	421555
2	USA	1992	447330	421555	447330
3	USA	1993	464757	421555	464757
4	USA	1994	512337	421555	512337
5	USA	1995	582965	421555	582965
6	USA	1996	622784	421555	622784
7	USA	1997	687533	421555	687533

The window frame
(w) is using default
settings of
UNBOUNDED
PRECEDING

I am then
calculating min and
max on w.

24.18 Using Window with Rows Between

```
SELECT
country,
year,
exports,
MIN(exports) OVER (w),
MAX(exports) OVER (w)
FROM trades
WHERE country = 'USA'
WINDOW w AS (ORDER BY year ROWS BETWEEN CURRENT ROW AND UNBOUNDED PRECEDING);
ERROR: frame end cannot be UNBOUNDED PRECEDING
LINE 9: ... w AS (ORDER BY year ROWS BETWEEN CURRENT ROW AND UNBOUNDED ...
                                                                    ^

SQL state: 42P20
Character: 167
```

Current row is coming before unbound preceding which is throwing an error.

```
SELECT
country,
year,
exports,
MIN(exports) OVER (w),
MAX(exports) OVER (w)
FROM trades
WHERE country = 'USA'
WINDOW w AS (ORDER BY year ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW);
ERROR: cannot copy window "w" because it has a frame clause
LINE 5: MIN(exports) OVER (w),
                          ^

HINT: Omit the parentheses in this OVER clause.
SQL state: 42P20
Character: 50
```

If I put unbound preceding before current row it says that it cannot use w with an over because it is in parenthesis


```

SELECT
country,
year,
exports,
MIN(exports) OVER w,
MAX(exports) OVER w
FROM trades
WHERE country = 'USA'
WINDOW w AS (ORDER BY year ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW);

```

If I omit the parenthesis when calling w in the over clause, now this fussy command works

	country text	year integer	exports numeric (50)	min numeric	max numeric
1	USA	1991	421555	421555	421555
2	USA	1992	447330	421555	447330
3	USA	1993	464757	421555	464757
4	USA	1994	512337	421555	512337
5	USA	1995	582965	421555	582965
6	USA	1996	622784	421555	622784
7	USA	1997	687533	421555	687533
8	USA	1998	680435	421555	687533
9	USA	1999	602784	421555	602784

I am using the window frame clause as a separate entity defined at the end of the SQL query. I am then calling the window frame twice with the over clause.

The beauty of this is that I only have to modify the window conditions in one place. For example I may want two rows before and 2 rows after as my window or 10 rows before and none after etc..

24.19 Rank & Dense_Rank functions

RANK() function returns the number of the current rows within it's window. Counting starts at 1.

The rank column will number tuples in the data set.

For RANK() based on the data, you may encounter duplicate ranks.

To avoid duplicate ranks we use DENSE_RANK().

Get the top 10 exports
by year for the USA

```
SELECT
year,
exports,
RANK() OVER (ORDER BY exports DESC) AS r
FROM trades
WHERE country = 'USA'
LIMIT 10;
```

	year integer	exports numeric (50)	r bigint
1	2018	1665303	1
2	2019	1644276	2
3	2014	1619743	3
4	2013	1577587	4
5	2017	1545810	5
6	2012	1544932	6
7	2015	1501846	7
8	2011	1481682	8
9	2016	1450906	9
10	2008	1299899	10

```
SELECT * FROM
(
SELECT
year,
exports,
RANK() OVER (ORDER BY exports DESC) AS r
FROM trades
WHERE country = 'USA'
LIMIT 10
) as T
WHERE r = 5;
```

	year integer	exports numeric (50)	r bigint
1	2017	1545810	5

Now if I push the whole query into a sub query and alias the output as T, I can do selections of the rank data with filtering using where etc..

24.20 Ntile Function

NTILE will split data into ideally equal groups

Divide ordered rows in the partition into a specific number of ranked buckets.

Bucket number for each group starts with 1

```
NTILE(buckets) OVER (  
[PARTITION BY partition_expression, ...]  
[ORDER BY sort_expression [ASC|DSC], ...])
```

Buckets represents the number of ranked groups and cannot be null

If the number of rows is not divisible by the buckets, function returns groups of two sizes

Get the USA exports from
year 2000 into 4 buckets

```
SELECT  
year,  
exports,  
NTILE(4) OVER (ORDER BY exports)  
FROM trades  
WHERE country = 'USA'  
AND year > 2000;
```

	year integer	exports numeric (50)	ntile integer
3	2001	729080	1
4	2004	814844	1
5	2005	901041	1
6	2006	1037029	2
7	2009	1056712	2
8	2007	1162538	2
9	2010	1278099	2
10	2008	1299899	2
11	2016	1450906	3
12	2011	1481682	3
13	2015	1501846	3
14	2012	1544932	3
15	2017	1545810	3
16	2013	1577587	4
17	2014	1619743	4
18	2019	1644276	4
19	2018	1665303	4

Note How I am asking for 4 buckets from 19 rows so the first bucket has 5 rows, the second bucket has 5 rows, the third 5 rows and the last has 4 rows. $5 + 5 + 5 + 4 = 19$.

```
SELECT
country,
year,
exports,
NTILE(5) OVER (PARTITION BY country ORDER BY exports)
FROM trades
WHERE country IN ('USA','France','Belgium')
AND year > 1995;
```

I am getting all rows where year is after 1995 for the three countries. I am then partitioning the data by country and ordering it by exports. Finally, the NTILE is then dividing EACH PARTITION into equally sized buckets.

	country text	year integer	exports numeric (50)	ntile integer
5	France	1998	300571	1
6	USA	1998	680435	2
7	Belgium	1999	178867	1
8	France	1999	296025	2
9	USA	1999	692784	3
10	Belgium	2000	187839	1
11	France	2000	295345	2
12	USA	2000	781831	3
13	Belgium	2001	190309	1
14	France	2001	289599	2
15	USA	2001	729080	3

I am partitioning by year but as there are only three countries the, buckets will be 1, 2 and 3, or if there is data for only two countries that year, 1 & 2.

```
SELECT
country,
year,
exports,
NTILE(5) OVER (PARTITION BY year ORDER BY exports)
FROM trades
WHERE country IN ('USA','France','Belgium')
AND year > 1995;
```

	country text	year integer	exports numeric (50)	ntile integer
17	Belgium	2018	468643	4
18	Belgium	2008	471798	5
19	Belgium	2014	472201	5
20	Belgium	2011	475958	5
21	Belgium	2013	511505	5
22	France	1997	283346	1
23	France	1996	283901	1
24	France	2001	289599	1
25	France	2000	295345	1
26	France	1999	296025	1
27	France	1998	300571	2
28	France	2002	304892	2
29	France	2003	358132	2
30	France	2004	413708	2
31	France	2005	434354	2
32	France	2009	464113	3
33	France	2006	479013	3

```

SELECT * FROM
(
SELECT
country,
year,
exports,
NTILE(5) OVER (PARTITION BY year ORDER BY exports) AS r
FROM trades
WHERE country IN ('USA','France','Belgium')
AND year > 1995
) as T
WHERE r = 3;

```

I can push the whole thing into a subqurt then extract only the rows with an NTILE bucket number of 3.

	country text	year integer	exports numeric (50)	r integer
1	USA	1999	692784	3
2	USA	2000	781831	3
3	USA	2001	729080	3
4	USA	2002	693068	3
5	USA	2003	724737	3
6	USA	2004	814844	3
7	USA	2005	901041	3
8	USA	2006	1037029	3
9	USA	2007	1162538	3
10	USA	2008	1299899	3
11	USA	2009	1056712	3

24.21 Lead & Lag Functions

LEAD and LAG functions allow you to move lines within the result set.

Very useful function to compare data of CURRENT ROW with any other rows (going backwards LAG and forwards LEAD).

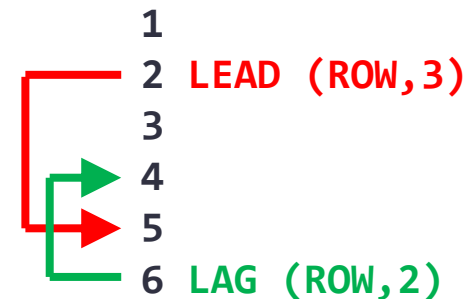
LEAD function access a row that follows the current row, at a specific offset.

After the current row

LEAD (expression, offset [,default_value])

If I am on row two and I want the lead with an offset of 3 rows then it will count 3 rows up from row 2 which will be 5.

If I am on row 6 and I want the lag with an offset of 2 then it will count two rows down from from row 6



Compare the exports
for a country year over
year.

```
SELECT  
year,  
exports,  
LEAD(exports,1) OVER (order BY year)  
FROM trades  
WHERE country = 'Belgium';
```

On the first row of the lead column it is getting the exports from the following year (2000). On the second row it is getting the exports from 2001 etc.

	year integer	exports numeric (50)	lead numeric
1	1999	178867	187839
2	2000	187839	190309
3	2001	190309	215803
4	2002	215803	255554
5	2003	255554	307690
6	2004	307690	335692
7	2005	335692	366835
8	2006	366835	431744
9	2007	431744	471798

On the first row of the lead column it is getting the exports from two years ahead (2001). On the second row it is getting the exports from 2002 etc.

```
SELECT
year,
exports,
LEAD(exports,2) OVER (order BY year)
FROM trades
WHERE country = 'Belgium';
```

	year integer	exports numeric (50)	lead numeric
1	1999	178867	190309
2	2000	187839	215803
3	2001	190309	255554
4	2002	215803	307690
5	2003	255554	335692
6	2004	307690	366835
7	2005	335692	431744
8	2006	366835	471798

	year integer	exports numeric (50)	lead numeric
1	2019	445214	430092
2	2018	468643	398306
3	2017	430092	397739
4	2016	398306	472201
5	2015	397739	511505
6	2014	472201	446854
7	2013	511505	475958

I can change the order by to descending but the lead function still works the same

```
SELECT
year,
exports,
LEAD(exports,2) OVER (order BY year DESC)
FROM trades
WHERE country = 'Belgium';
```

```
SELECT
year,
exports,
LAG(exports,1) OVER (order BY year)
FROM trades
WHERE country = 'Belgium';
```

The lag function works in reverse, taking the values from the offset rows before. In this case my offset is one so on the first column the value is null because there is no row before.

	year integer	exports numeric (50)	lag numeric
1	1999	178867	[null]
2	2000	187839	178867
3	2001	190309	187839
4	2002	215803	190309
5	2003	255554	215803

```
SELECT
country,
year,
exports,
LAG(exports,1) OVER (PARTITION BY country ORDER BY year)
FROM trades
WHERE country IN('USA','Belgium')
AND year > 2010;
```

I can use windowing on multiple countries where each partition is for a country.

The lag is working on each partition which is why the first lag value for each country is null.

	country text	year integer	exports numeric (50)	lag numeric
1	Belgium	2011	475958	[null]
2	Belgium	2012	446854	475958
3	Belgium	2013	511505	446854
4	Belgium	2014	472201	511505
5	Belgium	2015	397739	472201
6	Belgium	2016	398306	397739
7	Belgium	2017	430092	398306
8	Belgium	2018	468643	430092
9	Belgium	2019	445214	468643
10	USA	2011	1481682	[null]
11	USA	2012	1544932	1481682
12	USA	2013	1577587	1544932
13	USA	2014	1619743	1577587
14	USA	2015	1501846	1619743
15	USA	2016	1450906	1501846

If I order by year then it is comparing belgiums exports with USA exports on a yearly basis.

```
SELECT
country,
year,
exports,
LAG(exports,1) OVER (PARTITION BY year ORDER BY year)
FROM trades
WHERE country IN('USA','Belgium')
AND year > 2010;
```

	country text	year integer	exports numeric (50)	lag numeric
1	Belgium	2011	475958	[null]
2	USA	2011	1481682	475958
3	Belgium	2012	446854	[null]
4	USA	2012	1544932	446854
5	Belgium	2013	511505	[null]
6	USA	2013	1577587	511505
7	Belgium	2014	472201	[null]

24.22 First Value, Last Value & Nth Value

FIRST_VALUE() Returns the first value in a sorted partition of a result set.

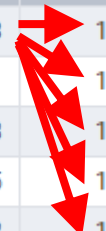
LAST_VALUE() Returns the last value in a sorted partition of a result set.

NTH_VALUE() Returns the value from the NTH ROW of a result set.

```
SELECT
year,
imports,
FIRST_VALUE(imports) OVER (ORDER BY year)
FROM trades
WHERE country = 'Belgium';
```

The FIRST_VALUE() function populates each row of the window with a column containing the first value of the entire result set.

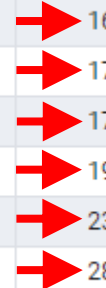
	year integer	imports numeric (50)	first_value numeric
1	1999	164603	164603
2	2000	176991	164603
3	2001	178698	164603
4	2002	198095	164603
5	2003	234922	164603
6	2004	286478	164603



```
SELECT
year,
imports,
LAST_VALUE(imports) OVER (ORDER BY year)
FROM trades
WHERE country = 'Belgium';
```

The expected outcome of LAST_VALUE() function would be that it populates the column with the last value from the result set but it is actually not doing that. For each row the last value is the same.

	year integer	imports numeric (50)	last_value numeric
1	1999	164603	164603
2	2000	176991	176991
3	2001	178698	178698
4	2002	198095	198095
5	2003	234922	234922
6	2004	286478	286478
7	2005	319085	319085
8	2006	353094	353094



```
SELECT
year,
imports,
LAST_VALUE(imports) OVER (ORDER BY year ROWS BETWEEN
                           0 PRECEDING AND
                           0 FOLLOWING)
FROM trades
WHERE country = 'Belgium';
```

The default behaviour of LAST_VALUE() is actually getting rows between 0 preceding and 0 following.

	year integer	imports numeric (50)	last_value numeric
1	1999	164603	164603
2	2000	176991	176991
3	2001	178698	178698
4	2002	198095	198095
5	2003	234922	234922
6	2004	286478	286478
7	2005	310085	310085

```
SELECT
year,
imports,
LAST_VALUE(imports) OVER (ORDER BY year ROWS BETWEEN
                           UNBOUNDED PRECEDING AND
                           UNBOUNDED FOLLOWING)
FROM trades
WHERE country = 'Belgium';
```

To get the LAST_VALUE() I need to take the rows between unbounded preceding and unbounded following.

	year integer	imports numeric (50)	last_value numeric
17	2015	371025	426489
18	2016	379298	426489
19	2017	409025	426489
20	2018	454714	426489
21	2019	426489	426489

If I order by country without specifying the bounds of the over clause...

```
SELECT
year,
imports,
LAST_VALUE(imports) OVER (ORDER BY country)
FROM trades
WHERE country = 'Belgium';
```

I get the same result where the LAST_VALUE() is actually the last value of the data set.

	year integer	imports numeric (50)	last_value numeric
17	2015	371025	426489
18	2016	379298	426489
19	2017	409025	426489
20	2018	454714	426489
21	2019	426489	426489

The Nth_value() will get a row value that I specify. For example I am populating the third column with the second value of imports from the data set

```
SELECT
year,
imports,
NTH_VALUE(imports, 2) OVER (ORDER BY year)
FROM trades
WHERE country = 'Belgium';
```

	year integer	imports numeric (50)	nth_value numeric
1	1999	164603	[null]
2	2000	176991	176991
3	2001	178698	176991
4	2002	198095	176991
5	2003	234922	176991

The Nth_value() will get a row value that I specify. For example I am populating the third column with the 4th value of imports from the data set

```
SELECT
year,
imports,
NTH_VALUE(imports, 4) OVER (ORDER BY year)
FROM trades
WHERE country = 'Belgium';
```

	year integer	imports numeric (50)	nth_value numeric
1	1999	164603	[null]
2	2000	176991	[null]
3	2001	178698	[null]
4	2002	198095	198095
5	2003	234922	198095
6	2004	286478	198095
7	2005	310025	198095

Note how the values are NULL until I get to the column where I want to take data.

```
SELECT
country,
year,
imports,
FIRST_VALUE(imports) OVER (ORDER BY year)
FROM trades
WHERE country IN ('Belgium', 'USA')
AND year > 2014;
```

Here I am picking multiple countries but without any partition so the first_value() is being populated in all rows.

	country text	year integer	imports numeric (50)	first_value numeric
1	Belgium	2015	371025	371025
2	USA	2015	2313425	371025
3	Belgium	2016	379298	371025
4	USA	2016	2247167	371025
5	Belgium	2017	409025	371025
6	USA	2017	2405277	371025
7	USA	2018	2611432	371025
8	Belgium	2018	454714	371025
9	Belgium	2019	426489	371025
10	USA	2019	2567492	371025

```
SELECT
country,
year,
imports,
FIRST_VALUE(imports) OVER (PARTITION BY country
                           ORDER BY year)
FROM trades
WHERE country IN ('Belgium', 'USA')
AND year > 2014;
```

If I partition the window by country then each partition will have a different first value.

	country text	year integer	imports numeric (50)	first_value numeric
1	Belgium	2015	371025	371025
2	Belgium	2016	379298	371025
3	Belgium	2017	409025	371025
4	Belgium	2018	454714	371025
5	Belgium	2019	426489	371025
6	USA	2015	2313425	2313425
7	USA	2016	2247167	2313425
8	USA	2017	2405277	2313425
9	USA	2018	2611432	2313425
10	USA	2019	2567492	2313425

```
SELECT
country,
year,
imports,
LAST_VALUE(imports) OVER (PARTITION BY country
                           ORDER BY year)
FROM trades
WHERE country IN ('Belgium', 'USA')
AND year > 2014;
```

Note how the last value is giving per row results because I am ordering by year.

	country text	year integer	imports numeric (50)	last_value numeric
1	Belgium	2015	371025	371025
2	Belgium	2016	379298	379298
3	Belgium	2017	409025	409025
4	Belgium	2018	454714	454714
5	Belgium	2019	426489	426489
6	USA	2015	2313425	2313425
7	USA	2016	2247167	2247167
8	USA	2017	2405277	2405277
9	USA	2018	2611432	2611432
10	USA	2019	2567492	2567492

```
SELECT
country,
year,
imports,
LAST_VALUE(imports) OVER (PARTITION BY country
                           ORDER BY country)
FROM trades
WHERE country IN ('Belgium', 'USA')
AND year > 2014;
```

If I order by country I get the last value for each partition.

	country text	year integer	imports numeric (50)	last_value numeric
1	Belgium	2015	371025	426489
2	Belgium	2016	379298	426489
3	Belgium	2017	409025	426489
4	Belgium	2018	454714	426489
5	Belgium	2019	426489	426489
6	USA	2015	2313425	2567492
7	USA	2016	2247167	2567492
8	USA	2017	2405277	2567492
9	USA	2018	2611432	2567492
10	USA	2019	2567492	2567492

24.23 Row_Number Function

Can simply be used to return a virtual ID.

Assign a unique integer value to each row in a result set starting from 1.

```
SELECT
year,
imports,
ROW_NUMBER() OVER (ORDER BY imports)
FROM trades
WHERE country = 'France';
```

```
SELECT
year,
imports,
ROW_NUMBER() OVER (ORDER BY imports DESC)
FROM trades
WHERE country = 'France';
```

```
SELECT
Country,
year,
imports,
ROW_NUMBER() OVER (PARTITION BY country
ORDER BY imports DESC)
FROM trades
```

Each partition by country gets a row number.

	year integer	imports numeric (50)	row_number bigint
1	1994	228282	1
2	1997	266575	2
3	1995	273387	3
4	1996	274914	4
5	1998	286036	5

	year integer	imports numeric (50)	row_number bigint
1	2011	713675	1
2	2008	695004	2
3	2013	671254	3
4	2012	666675	4
5	2014	659872	5

	country text	year integer	imports numeric (50)	row_number bigint
9	Afghanistan	2009	3336	9
10	Afghanistan	2008	3020	10
11	Albania	2018	5941	1
12	Albania	2011	5396	2
13	Albania	2017	5294	3
14	Albania	2008	5250	4
15	Albania	2014	5230	5
16	Albania	2013	4881	6

```

SELECT * FROM
(
  SELECT
  country,
  year,
  imports,
  ROW_NUMBER() OVER (PARTITION BY country
                      ORDER BY imports DESC)
  FROM trades
) AS t
WHERE row_number = 4;

```

	country text	year integer	imports numeric (50)	row_number bigint
1	Afghanistan	2018	7407	4
2	Albania	2008	5250	4
3	Algeria	2012	50369	4
4	Andorra	2004	1762	4
5	Angola	2009	23919	4
6	Anguilla	2004	102	4
7	Antigua and Barbuda	2007	573	4

I am putting the whole window partitioning query within a sub query. The window partitioning query is first partitioning the data by country then ordering each window partition by exports descending, i.e. lowest to highest imports.

In the outer query I am filtering the inner query to get the 4th row for each country, i.e. I am finding the 4th lowest import year for each country.

```

SELECT * FROM
(
  SELECT
  country,
  year,
  imports,
  ROW_NUMBER() OVER (PARTITION BY country
                      ORDER BY year)
  FROM trades
  WHERE YEAR >= 2010
) AS t
WHERE row_number = 4
ORDER BY imports DESC;

```

	country text	year integer	imports numeric (50)	row_number bigint
1	USA	2013	2326590	4
2	China	2013	1949992	4
3	Germany	2013	1187302	4
4	Japan	2013	833166	4
5	France	2013	671254	4
6	United Kingdom	2013	657223	4

If I order by year and select the year from 2010 then the 4th row on the outer query would be the year 2013.

24.24 Finding Correlations

The correlation function, `corr()` can be used to determine how two numbers relate to each other.

The correlation function calculates a correlation coefficient with values from -1 to 1

Correlation of 1 indicates the strongest correlation and -1 is the strongest inverse correlation.

```
SELECT
country,
corr(imports, exports)
FROM trades
GROUP BY country
ORDER BY 2 DESC
NULLS LAST;
```

Looking at the correlation between imports and exports for each country, I see that NL, Switzerland, Germany etc have a high positive correlation. That is to say when their imports go up, their exports go up.

	country text	corr double precision
1	French Guiana	1
2	Martinique	1
3	Guadeloupe	1
4	Fmr Fed. Rep. of Germany	0.9995397437431122
5	Netherlands	0.9990556444188695
6	Switzerland	0.9988720548158271
7	Belgium	0.9988289179592891
8	Mexico	0.9988262381611238
9	Slovakia	0.9985798332503386
10	Germany	0.9977792067485522


```
SELECT
country,
corr(imports, exports)
FROM trades
WHERE country IN ('USA', 'France', 'Japan', 'Singapore')
GROUP BY country
ORDER BY 2 DESC
NULLS LAST;
```

	country text	corr double precision
1	France	0.9975115468299817
2	Singapore	0.9971660106833287
3	USA	0.9792447865033233
4	Japan	0.9733714502844296

The same for USA, France, Japan and Singapore. All these countries have a positive correlation between imports and exports.

```
SELECT
country,
corr(imports, exports)
FROM trades
GROUP BY country
ORDER BY 2 DESC
NULLS LAST;
```

At the other end of the scale these countries have a very negative correlation, that is to say as their imports go up, their exports go down.

	country text	corr double precision
186	Grenada	-0.0042194000400017
187	Libya	-0.01809700753506633
188	So. African Customs Union	-0.019787293521626975
189	Aruba	-0.021282485566721115
190	Saint Vincent and the Grenadines	-0.34482553676809913
191	Cook Isds	-0.4906926950624538
192	Samoa	-0.5117980501267959
193	Timor-Leste	-0.6143282131063926
194	China, Macao SAR	-0.6306078105831799
195	French Polynesia	-0.6343660970323639
196	Dominica	-0.6544811695808521
197	Bermuda	-0.6746955322773635
198	Guinea-Bissau	-0.9579251282612282

24.25 Row_Number() – Window with Partition Datasets

PostgreSQL ROW_NUMBER() function to assign a unique sequential integer value to each row in a result set.

What is a window? – A window is the set of rows on which the ROW_NUMBER() function operates.

```
ROW_NUMBER() OVER  
(  
PARTITION BY column_1, column_2  
ORDER BY column_3, column_4  
)
```

The PARTITION BY clause normally divides the window into smaller sets of partitions. If we specify the PARTITION BY clause, the row number for each partition starts with 1 and increments by +1.

PARTITION BY clause is optional

If we do not use the partition by clause then the ROW_NUMBER() function will treat the whole window as a single partition or one partition.

The ORDER BY determines the order in which the numbers are assigned.

Get first_name, last_name, salary, and department for all employees and then use ROW_NUMBER() to partition by department name and order by salary.

	first_name character varying (20)	last_name character varying (25)	salary numeric (8,2)	department_name character varying (30)	row_number bigint
1	TJ	Olson	2100.00	Shipping	1
2	Hazel	Philtanker	2200.00	Shipping	2
3	Steven	Markle	2200.00	Shipping	3
4	Ki	Gee	2400.00	Shipping	4
5	James	Landry	2400.00	Shipping	5
6	Peter	Vargas	2500.00	Shipping	6

```
SELECT
e.first_name,
e.last_name,
e.salary,
d.department_name,
ROW_NUMBER() OVER (ORDER BY e.salary)
FROM employees e
INNER JOIN departments d ON d.department_id = e.department_id;
```

Get first_name, last_name, salary, and department for all employees and then use ROW_NUMBER() to order by salary.

	first_name character varying (20)	last_name character varying (25)	salary numeric (8,2)	department_name character varying (30)	row_number bigint
1	William	Gietz	8300.00	Accounting	1
2	Shelley	Higgins	12000.00	Accounting	2
3	Jennifer	Whalen	4400.00	Administration	1
4	Lex	De Haan	17000.00	Executive	1
5	Neena	Kochhar	17000.00	Executive	2
6	Steven	King	24000.00	Executive	3
7	Luis	Popp	6900.00	Finance	1

```
SELECT
e.first_name,
e.last_name,
e.salary,
d.department_name,
ROW_NUMBER() OVER (PARTITION BY d.department_name ORDER BY e.salary)
FROM employees e
INNER JOIN departments d ON d.department_id = e.department_id;
```

24.26 Row_Number() – Reverse Fields with Order By

Reversing the order of salary gives highest to lowest salary in each partition (department)

```
SELECT
e.first_name,
e.last_name,
e.salary,
d.department_name,
ROW_NUMBER() OVER (PARTITION BY d.department_name ORDER BY e.salary DESC)
FROM employees e
INNER JOIN departments d ON d.department_id = e.department_id;
```

	first_name character varying (20)	last_name character varying (25)	salary numeric (8,2)	department_name character varying (30)	row_number bigint
1	Shelley	Higgins	12000.00	Accounting	1
2	William	Gietz	8300.00	Accounting	2
3	Jennifer	Whalen	4400.00	Administration	1
4	Steven	King	24000.00	Executive	1
5	Neena	Kochhar	17000.00	Executive	2
6	Lex	De Haan	17000.00	Executive	3
7	Nancy	Greenberg	12000.00	Finance	1

I can partition by salary then order by department.

```
SELECT
e.first_name,
e.last_name,
e.salary,
d.department_name,
ROW_NUMBER() OVER (PARTITION BY d.department_name ORDER BY e.salary DESC)
FROM employees e
INNER JOIN departments d ON d.department_id = e.department_id;
```

	first_name character varying (20)	last_name character varying (25)	salary numeric (8,2)	department_name character varying (30)	row_number bigint
53	Sundita	Kumar	6100.00	Sales	1
54	Charles	Johnson	6200.00	Sales	1
55	Amit	Banda	6200.00	Sales	2
56	Sundar	Ande	6400.00	Sales	1
57	Shanta	Vollman	6500.00	Shipping	1
58	Susan	Mavris	6500.00	Human Resources	2
59	David	Lee	6800.00	Sales	1
60	Luis	Popp	6900.00	Finance	1

24.27 Row_Number() – Find Nth Record

Select first_name, last_name, salary and department names for the 2nd highest paid employee for each department.

```
SELECT * FROM
(
  SELECT
  e.first_name,
  e.last_name,
  e.salary,
  d.department_name,
  ROW_NUMBER() OVER (PARTITION BY d.department_name ORDER BY e.salary DESC)
  FROM employees e
  INNER JOIN departments d ON d.department_id = e.department_id
) as t
WHERE row_number = 2;
```

	first_name character varying (20) 🔒	last_name character varying (25) 🔒	salary numeric (8,2) 🔒	department_name character varying (30) 🔒	row_number bigint 🔒
1	William	Gietz	8300.00	Accounting	2
2	Neena	Kochhar	17000.00	Executive	2
3	Daniel	Faviet	9000.00	Finance	2
4	Bruce	Ernst	6000.00	IT	2
5	Pat	Fay	6000.00	Marketing	2
6	Alexander	Khoo	3100.00	Purchasing	2
7	Karen	Partners	13500.00	Sales	2

24.28 Row_Number() – Find Distinct with Subquery

Get all the distinct salaries and assign ROW_NUMBER() on them.

```
SELECT DISTINCT
salary,
ROW_NUMBER() OVER (ORDER BY salary)
FROM employees
ORDER BY salary DESC;
```

Using the distinct keyword in the select is still producing duplicates because the ORDER BY clause of the ROW_NUMBER was performed before the DISTINCT of the SELECT.

	salary numeric (8,2) 🔒	row_number bigint 🔒
1	24000.00	107
2	17000.00	105
3	17000.00	106
4	14000.00	104
5	13500.00	103
6	13000.00	102
7	12000.00	101
8	12000.00	99
9	12000.00	100

To solve this I first select all distinct rows for salary in a sub query.

In the outer query I perform the row_number() OVER and (ORDER BY salary)

```
SELECT
salary,
ROW_NUMBER() OVER (ORDER BY salary
DESC)
FROM
(
SELECT DISTINCT
salary
FROM employees
) as t
```

What the system does is take the distinct rows from the inner query then apply the row_numbers on the outer query.

	salary numeric (8,2) 🔒	row_number bigint 🔒
29	6800.00	29
30	6500.00	30
31	6400.00	31
32	6200.00	32
33	6100.00	33
34	6000.00	34
35	5800.00	35
36	4800.00	36
37	4400.00	37

24.29 Row_Number() – Pagination Technique

Get firstname, last_name, salary & department for all employees and then use ROW_NUMBER() to partition by department name and order by salary. Select 5 rows starting at row 6.

	first_name character varying (20)	last_name character varying (25)	salary numeric (8,2)	department_name character varying (30)	row_number bigint
3	Sarath	Sewall	7000.00	Sales	6
4	Oliver	Tuvault	7000.00	Sales	7
5	Mattea	Marvins	7200.00	Sales	8
6	Elizabeth	Bates	7300.00	Sales	9
7	William	Smith	7400.00	Sales	10

```
SELECT * FROM
(
  SELECT
    e.first_name,
    e.last_name,
    e.salary,
    d.department_name,
    ROW_NUMBER() OVER (PARTITION BY d.department_name ORDER BY e.salary)
  FROM employees e
  INNER JOIN departments d ON d.department_id = e.department_id
) as t
WHERE row_number BETWEEN 6 AND 10;
```

```
SELECT * FROM
(
  SELECT
    salary,
    ROW_NUMBER() OVER (ORDER BY salary)
  FROM employees e
) as t
WHERE row_number BETWEEN 6 AND 10;
```

	salary numeric (8,2)	row_number bigint
1	2500.00	6
2	2500.00	7
3	2500.00	8
4	2500.00	9
5	2500.00	10

List all salaries
from row 6 to 10

24.30 Using over Function to Calculate Percentage

Employee salary percentage vs total

```
SELECT
first_name,
last_name,
salary,
SUM(salary) OVER ()
FROM employees
```

	first_name character varying (20)	last_name character varying (25)	salary numeric (8,2)	sum numeric
1	Steven	King	24000.00	691400.00
2	Neena	Kochhar	17000.00	691400.00
3	Lex	De Haan	17000.00	691400.00
4	Alexander	Hunold	9000.00	691400.00
5	Baer	Frost	6000.00	691400.00

For the entire dataset, i.e. one partition, the total salary for all employees is 691400

```
SELECT
first_name,
last_name,
salary,
SUM(salary) OVER () AS total_salary,
round(salary / SUM(salary) OVER () * 100,2) AS pct_salary
FROM employees
ORDER BY 5 DESC;
```

	first_name character varying (20)	last_name character varying (25)	salary numeric (8,2)	total_salary numeric	pct_salary numeric
1	Steven	King	24000.00	691400.00	3.47
2	Neena	Kochhar	17000.00	691400.00	2.46
3	Lex	De Haan	17000.00	691400.00	2.46
4	John	Russell	14000.00	691400.00	2.02
5	Karen	Partners	13500.00	691400.00	1.95
6	Michael	Hartstein	13000.00	691400.00	1.88

Steven is paid 3.47% of the total salary for all employees.

24.31 Calculate Difference Compared to Average

```
SELECT
first_name,
last_name,
salary,
round(Salary - AVG(salary) OVER (),2) AS salary_less_avg,
round(AVG(salary) OVER (),2) AS total_avg_salary
FROM employees
ORDER BY 5 DESC;
```

	first_name character varying (20)	last_name character varying (25)	salary numeric (8,2)	salary_less_avg numeric	total_avg_salary numeric
1	Steven	King	24000.00	17538.32	6461.68
2	Neena	Kochhar	17000.00	10538.32	6461.68
3	Lex	De Haan	17000.00	10538.32	6461.68
4	Alexander	Hunold	9000.00	2538.32	6461.68
5	Bruce	Ernst	6000.00	-461.68	6461.68
6	David	Austin	4800.00	-1661.68	6461.68

Steven is paid 17538.32 more than the average salary of all employees whereas Bruce is paid 461.68 less than the average salary of all employees.

```
SELECT
first_name,
last_name,
salary,
round(Salary - AVG(salary) OVER (),2) AS salary_less_avg,
round(AVG(salary) OVER (),2) AS total_avg_salary
FROM employees
ORDER BY 4 DESC;
```

This query could be further refined to order by column 4 to order from highest above average salary to lowest below average salary.

24.32 Calculating Cumulative Totals

```
SELECT
first_name,
last_name,
salary,
SUM(salary) OVER (),
SUM(salary) OVER (ORDER BY salary DESC ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
AS cum_total
FROM employees;
```

To get a cumulative total I want to sum all the preceding rows (UNBOUNDED PRECEDING) and the current row.

	first_name character varying (20) 🔒	last_name character varying (25) 🔒	salary numeric (8,2) 🔒	sum numeric 🔒	cum_total numeric 🔒	
1	Steven	King	24000.00	~~691400.00~~	24000.00	1
2	Neena	Kochhar	17000.00	~~691400.00~~	41000.00	2
3	Lex	De Haan	17000.00	~~691400.00~~	58000.00	3
4	John	Russell	14000.00	691400.00	72000.00	
5	Karen	Partners	13500.00	691400.00	85500.00	
6	Michael	Hartstein	13000.00	691400.00	98500.00	
7	Shelley	Higgins	12000.00	691400.00	110500.00	
8	Nancy	Greenberg	12000.00	691400.00	122500.00	

$$1 \text{ } 2400 + 0 = 2400$$

$$1 \text{ } 2400 + 1700 = 4100$$

$$1 \text{ } 2400 + 1700 + 1700 = 5800$$

24.33 Using Lead to Compare with Next Values

```
SELECT
first_name,
last_name,
salary,
salary - LEAD(salary, 1) OVER (ORDER BY salary DESC) AS sal_diff_next
FROM employees
ORDER BY salary DESC;
```

	first_name character varying (20) 🔒	last_name character varying (25) 🔒	salary numeric (8,2) 🔒	sal_diff_next numeric 🔒
1	Steven	King	24000.00	7000.00
2	Neena	Kochhar	17000.00	0.00
3	Lex	De Haan	17000.00	3000.00
4	John	Russell	14000.00	500.00
5	Karen	Partners	13500.00	500.00
6	Michael	Hartstein	13000.00	1000.00
7	Shelley	Higgins	12000.00	0.00
8	Nancy	Greenberg	12000.00	0.00

The salary difference between Steven and Neena is 7000.

The salary difference between Neena and Lex is zero

The salary difference Lex and John is 3000

```
SELECT
first_name,
last_name,
salary,
salary - LAG(salary, 1) OVER (ORDER BY salary DESC) AS sal_diff_prev
FROM employees
ORDER BY salary DESC;
```

	first_name character varying (20) 🔒	last_name character varying (25) 🔒	salary numeric (8,2) 🔒	sal_diff_prev numeric 🔒
1	Steven	King	24000.00	→ [null]
2	Neena	Kochhar	17000.00	→ -7000.00
3	Lex	De Haan	17000.00	→ 0.00
4	John	Russell	14000.00	-3000.00
5	Karen	Partners	13500.00	-500.00
6	Michael	Hartstein	13000.00	-500.00
7	Shelley	Higgins	12000.00	-1000.00

The salary difference between steven and the previous row is NULL because there is no previous row

The salary difference between Neena and steven is 7000

The salary difference between Lex and Nena is zero

24.34 Comparing with Least number

```
SELECT
salary,
salary - LAST_VALUE(salary) OVER w AS sal_v_lowest,
salary - FIRST_VALUE(salary) OVER w AS sal_v_higest
FROM employees
WINDOW w AS (ORDER BY salary DESC ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING);
```

	first_name character varying (20) 🔒	last_name character varying (25) 🔒	salary numeric (8,2) 🔒	sal_v_lowest numeric 🔒	sal_v_higest numeric 🔒	
1	Steven	King	24000.00	21900.00	0.00	1
2	Neena	Kochhar	17000.00	14900.00	-7000.00	2
3	Lex	De Haan	17000.00	14900.00	-7000.00	3
4	John	Russell	14000.00	11900.00	-10000.00	
5	Karen	Partners	13500.00	11400.00	-10500.00	
6	Michael	Hartstein	13000.00	10900.00	-11000.00	

The salary difference for Steven between the lowest paid is 21900 and between the highest paid is 0 because Steven is the highest paid.

The Salary difference between Neena and the lowest paid is 14900 and between Neena and the highest paid is -7000

The salary difference between Lex and the lowest paid is 14900 and between Lex and the highest paid is -7000

```

SELECT
first_name,
last_name,
salary,
salary - LAST_VALUE(salary) OVER w AS sal_v_lowest,
ROUND((salary - LAST_VALUE(salary) OVER w) / LAST_VALUE(salary) OVER w * 100,2) AS pct_v_lowest,
salary - FIRST_VALUE(salary) OVER w AS sal_v_highest,
ROUND((salary - FIRST_VALUE(salary) OVER w) / FIRST_VALUE(salary) OVER w * 100,2) AS pct_v_highest
FROM employees
WINDOW w AS (ORDER BY salary DESC ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)
ORDER BY salary DESC;

```

	first_name character varying (20) 🔒	last_name character varying (25) 🔒	salary numeric (8,2) 🔒	sal_v_lowest numeric 🔒	pct_v_lowest numeric 🔒	sal_v_highest numeric 🔒	pct_v_highest numeric 🔒	
1	Steven	King	24000.00	21900.00	1042.86	0.00	0.00	1
2	Neena	Kochhar	17000.00	14900.00	709.52	-7000.00	-29.17	2
3	Lex	De Haan	17000.00	14900.00	709.52	-7000.00	-29.17	
4	John	Russell	14000.00	11900.00	566.67	-10000.00	-41.67	
5	Karen	Partners	13500.00	11400.00	542.86	-10500.00	-43.75	
6	Michael	Hartstein	13000.00	10900.00	519.05	-11000.00	-45.83	3
7	Shelley	Higgins	12000.00	9000.00	471.43	-12000.00	-50.00	

Steven is getting paid 21900 more than the lowest paid person which is a 1042.86% increase.

Neena is getting paid 14900 more than the lowest paid person which is a 709.52% increase.
 Neena's salary is 7000 less than the highest paid person or 29.17% less than the highest paid person

Michael is getting paid 10900 more than the lowest paid person which is a 519% increase.
 Michael is getting paid 11000 less than the highest paid person which is 45.83% less.

24.35 Window Function Summary

Scope	Type	Function	Description
Frame	Computation	generic aggs	e.g. SUM, AVG
	row access	FIRST_VALUE LAST_VALUE NTH_VALUE	First frame value Last Frame value Nth frame value
	row access	LAG LEAD ROW_NUMBER	Row before current Row after current Current row number
Partition	ranking	CUME_DIST DENSE_RANK NTILE PERCENT_RANK RANK	Cumulative distribution Rank without gaps Rank in n partitions Percent in rank Rank with gaps

Windows functions never process rows outside their partitions.
However, without PARTITION BY, the partition is the entire set.

24.36 Planning Tips for Using Windows Functions

Function_name OVER (PARTITION BY ... ORDER BY ... frame_clause)

- Do you want to split/PARTITION the dataset (PARTITION BY creates multiple sets/partitions)
- Do you want to ORDER in the partition? (use ORDER BY)
- How do you want to handle ROWS with the same ORDER BY values?
 - RANGE vs ROWS
 - RANK vs DENSE RANK
- Do you need to define WINDOW FRAME (ROWS BETWEEN ...)
- Window functions can handle their own:-
 - Partitions
 - ordering
 - and window frames
- Multiple window names can be defined in the WINDOW clause
- Pay attention to whether window functions operate on:-
 - Frames or
 - Partitions

24.37 Difference between Rank & Dense_Rank

```
SELECT
first_name,
last_name,
salary,
RANK() OVER w
FROM employees
WINDOW w AS (ORDER BY salary DESC);
```

	first_name character varying (20)	last_name character varying (25)	salary numeric (8,2)	rank bigint
1	Steven	King	24000.00	1
2	Neena	Kochhar	17000.00	2
3	Lex	De Haan	17000.00	2
4	John	Russell	14000.00	4
5	Karen	Partners	13500.00	5
6	Michael	Hartstein	13000.00	6
7	Shelley	Higgins	12000.00	7
8	Nancy	Greenberg	12000.00	7

```
SELECT
first_name,
last_name,
salary,
DENSE_RANK() OVER w
FROM employees
WINDOW w AS (ORDER BY salary DESC);
```

	first_name character varying (20)	last_name character varying (25)	salary numeric (8,2)	dense_rank bigint
1	Steven	King	24000.00	1
2	Neena	Kochhar	17000.00	2
3	Lex	De Haan	17000.00	2
4	John	Russell	14000.00	3
5	Karen	Partners	13500.00	4
6	Michael	Hartstein	13000.00	5
7	Shelley	Higgins	12000.00	6
8	Nancy	Greenberg	12000.00	6

Note how rank is numbering the rows in the window and giving the same rank number to equal values. Rank will skip a value if there are same values so in this case there are two rows with a rank of 2 then it skips to 4. Dense rank does the same as rank but without gaps i.e. same values have the same rank but it does not skip a rank value.

24.38 Getting Rank to compute group and Global Ranks

Rank over department_id and order by salary. Each department has the same rank value

```
SELECT
first_name,
salary,
department_id,
RANK() OVER w_department
FROM employees
WINDOW w_department AS (PARTITION BY department_id ORDER BY salary DESC)
ORDER BY department_id, salary DESC
```

	first_name character varying (20)	salary numeric (8,2)	department_id numeric (4)	rank bigint
1	Kimberely	7000.00	0	1
2	Jennifer	4400.00	10	1
3	Michael	13000.00	20	1
4	Pat	6000.00	20	2
5	Den	11000.00	30	1
6	Alexander	3100.00	30	2
7	Shelli	2900.00	30	3

Compute rank within department partition and overall rank

```
SELECT
first_name,
salary,
department_id,
RANK() OVER w_department,
RANK() OVER w_all_departments
FROM employees
WINDOW
w_department AS (PARTITION BY department_id ORDER BY salary DESC),
w_all_departments AS (ORDER BY salary DESC)
ORDER BY department_id, salary DESC
```

	first_name character varying (20)	salary numeric (8,2)	department_id numeric (4)	rank bigint	rank bigint
1	Kimberely	7000.00	0	1	45
2	Jennifer	4400.00	10	1	61
3	Michael	13000.00	20	1	6
4	Pat	6000.00	20	2	56
5	Den	11000.00	30	1	11
6	Alexander	3100.00	30	2	78
7	Shelli	2900.00	30	3	84

24.39 Using Partition By for Group Averages

```
SELECT
first_name,
salary,
department_id,
ROUND(AVG(salary) OVER (PARTITION BY department_id),2) AS avg,
ROUND(salary - AVG(salary) OVER (PARTITION BY department_id),2) AS
diff_avg
FROM employees
```

	first_name character varying (20) 🔒	salary numeric (8,2) 🔒	department_id numeric (4) 🔒	avg numeric 🔒	diff_avg numeric 🔒
1	Kimberely	7000.00	0	7000.00	0.00
2	Jennifer	4400.00	10	4400.00	0.00
3	Pat	6000.00	20	9500.00	-3500.00
4	Michael	13000.00	20	9500.00	3500.00
5	Den	11000.00	30	4150.00	6850.00
6	Shelli	2900.00	30	4150.00	-1250.00
7	Guy	2600.00	30	4150.00	-1550.00

Kimberly is in department zero which has an average salary of 7000 which is also the average overall salary for the whole department.

Pat is in department 20, earning 6000 where the average department salary is 9500 so he is earning 3500 below the average department salary.

24.40 Using With Clause to Create Own Data

```
WITH num (mynum) AS  
(  
VALUES (1),(2),(3),(4)  
)  
SELECT * FROM num;
```

	mynum integer
1	1
2	2
3	3
4	4

```
WITH num (mynum) AS  
(  
VALUES  
(1),(2),(3),(4)  
)  
SELECT  
MIN(mynum),  
AVG(mynum)  
FROM num;
```

	min integer	avg numeric
1	1	2.5000000000000000

I perform operations
on data that I create.

I can create my own data in a select
sub query with a list of values then
call that data from an outer query.

24.41 Using With Clause to set Order By Values

```
CREATE TABLE t_order (  
id SERIAL PRIMARY KEY,  
item VARCHAR(100)  
);
```

```
INSERT INTO t_order (item) VALUES  
(  
'Pen'),  
(  
'Paper'),  
(  
'Pencil'),  
(  
'Book'),  
(  
'Printer');
```

```
SELECT * FROM t_order  
ORDER BY item DESC
```

	id [PK] integer	item character varying (100)
1	5	Printer
2	3	Pencil
3	1	Pen
4	2	Paper
5	4	Book

But what if I don't want to order by the columns but order by a
set of values that I define?

```

WITH t_data (list_order, item) AS
(
VALUES
(1, 'Book'),
(2, 'Pencil'),
(3, 'Pen'),
(4, 'Printer'),
(5, 'Paper')
)
SELECT
*
FROM t_order
JOIN t_data ON t_data.item = t_order.item

```

	id [PK] integer	item character varying (100)
1	1	Pen
2	2	Paper
3	3	Pencil
4	4	Book
5	5	Printer

	list_order integer	item text
	3	Pen
	5	Paper
	2	Pencil
	1	Book
	4	Printer

Inside the values of the WITH is the order that I want the values to appear in.

I can select from the original order and join it with the temporary table, 't_data'

```

WITH t_data (list_order, item) AS
(
VALUES
(1, 'Book'),
(2, 'Pencil'),
(3, 'Pen'),
(4, 'Printer'),
(5, 'Paper')
)
SELECT
*
FROM t_order
JOIN t_data ON t_data.item = t_order.item
ORDER BY list_order

```

	id [PK] integer	item character varying (100)	list_order integer	item text
1	4	Book	1	Book
2	3	Pencil	2	Pencil
3	1	Pen	3	Pen
4	5	Printer	4	Printer
5	2	Paper	5	Paper

I can use an order by to order not by the values in the 't_order' table but by a column from the temporary table.

24.42 Delete and Insert in One query using With

```
CREATE TABLE t1 (  
    id SERIAL PRIMARY KEY,  
    item VARCHAR(10)  
);
```

```
CREATE TABLE t2 (  
    id SERIAL PRIMARY KEY,  
    item VARCHAR(10)  
);
```

```
INSERT INTO t1 (item)  
VALUES  
('item1'),  
('item2'),  
('item2'),  
('item2');
```

```
WITH yahoo AS  
(  
    DELETE FROM t1  
    WHERE id IN (1,2)  
    RETURNING item  
)  
INSERT INTO t2 (item)  
SELECT * FROM yahoo;
```

```
SELECT * FROM t1;
```

	id [PK] integer	item character varying (10)
1	3	item2
2	4	item2

```
SELECT * FROM t2;
```

	id [PK] integer	item character varying (10)
1	1	item1
2	2	item2

25. Using Regular Expressions for Text Patterns

[25.1 Text to Structured Data](#)

[25.2 Regular Expressions Notations Table](#)

[25.3 SIMILAR TO Operator](#)

[25.4 POSIX Regular Expressions](#)

[25.5 SUBSTRING with Regular Expressions](#)

[25.6 REGEXP MATCHES Function](#)

[25.7 REGEXP REPLACE Function](#)

[25.8 REGEXP SPLIT TO TABLE Function](#)

[25.9 REGEXP SPLIT TO ARRAY Function](#)

25.1 Text to Structured Data

Whether you are looking for data in text, you need to first transform it into a format you can use. PostgreSQL has more than 50 built-in string functions for that. Some are part of ANSI SQL standard and some are specific to PostgreSQL.

<https://www.postgresql.org/docs/9.1/functions-string.html>

25.2 Regular Expressions Notations Table

Regex or regular expressions are a type of notational language that describe 'text patterns'

Expression	Notes
.	A dot is a wildcard that finds any character except a new line
[FGz]	Any Character in the square bracket, here is F,G & z
[a-z]	A range of characters. Here it is lowercase a to z
[A-Z]	A range of characters. Her it is uppercase A to Z
[^a-z]	The caret negates the match. Here not lowercase a to z
\w	Any word, character or underscore. Same as [a-zA-Z0-9_]

Expression	Notes
\d	Any Digit
\t	Tab character
\s	A space
\n	New line character
\r	Character return
\r\n	New line or Character return
^	Match at the start of the string
\$	Match at the end of the string
?	Get the preceding match zero or one time
*	Get the preceding match zero or more times
+	Get the preceding match one or more times
{m}	Get the preceding match exactly m times
{m,n}	Get the preceding match between M and n times
A b	The pipe denotes alternation. Find ether a or b
()	Set precedence
(?:)	Negate the reporting of a capture group

25.3 SIMILAR TO Operator

The SIMILAR TO operator returns true or false depending on whether its pattern matches the given string.

It is similar to LIKE except that it interprets the pattern using the standard's definition of a regular expression.

SQL regular expressions are a curious cross between LIKE notation and common regular expression notation.

Like LIKE, the SIMILAR TO operator succeeds only if its pattern matches the entire string; this is unlike common regular expression behaviour where the pattern can match any part of the string.

Also like LIKE, SIMILAR TO uses `_` and `%` as wildcard characters denoting any single character and any string, respectively (these are comparable to `.` And `.*` in POSIX regular expressions)

```
SELECT
'same' SIMILAR TO 'same',
'Same' SIMILAR TO 'same', -- case sensitive matching
'same' SIMILAR TO 's';    -- partial character matching
```

	?column? boolean	?column? boolean	?column? boolean
1	true	false	false

```
SELECT
'same' SIMILAR TO 's%', -- wildcard after first char
'same' SIMILAR TO '%s', -- wildcard before first char
'same' SIMILAR TO 'sa%'; -- wildcard after 2nd char
```

	?column? boolean	?column? boolean	?column? boolean
1	true	false	true

```
SELECT  
'same' SIMILAR TO '%(s|a)%';  
-- | demotes alternation (either of two alternatives)
```

	?column? boolean 
1	true

'%s' would evaluate as false but 'a%' evaluates as true so as we are using a pipe to denote alternation, this will return true because one of them matches.

```
SELECT  
'same' SIMILAR TO '(m|e)%';  
-- | demotes alternation (either of two alternatives)
```

	?column? boolean 
1	false

'm' would evaluate as false and 'e%' evaluates as false so as we are using a pipe to denote alternation, this will return false because none of them matches.

25.4 POSIX Regular Expressions

A regular expression is a special text string used to describe a search pattern. PostgreSQL's regular expressions support three approaches to pattern matching:

POSIX-style regular expressions (BREs and EREs)

SIMILAR TO operator added in SQL:1999

SQL LIKE operator

POSIX regular expressions provide a more powerful means for pattern matching than LIKE and SIMILAR TO.

PostgreSQL supports following four operators for POSIX regular expression matching (also known as the tilde operator). The tilde operator returns true or false depending on whether or not a regular expression can match a string or part thereof.

~ (Matches regular expression, case sensitive)

~* (Matches regular expression, case insensitive)

!~ (Does not match regular expression, case sensitive)

!~* (Does not match regular expression, case insensitive)

```
SELECT
'same' ~ 'same' AS result1,
'same' ~ 'Same' AS result2,
'same' ~* 'Same' AS result3,
'same' !~ 'Same' AS result4,
'same' !~* 'Same' AS result5;
```

	result1 	result2 	result3 	result4 	result5 
1	true	false	true	true	false

25.5 SUBSTRING with Regular Expressions

```
SELECT
substring ('The movie will start at 8 p.m on Dec 10, 2020' FROM '.'),
substring ('The movie will start at 8 p.m on Dec 10, 2020' FROM '.*');
```

	substring text	substring text
1	T	The movie will start at 8 p.m on Dec 10, 2020

The dot will find any character, dot* will find any character repeated zero or more times

```
SELECT
substring ('The movie will start at 8 p.m on Dec 10, 2020' FROM 'movie.+'),
substring ('The movie will start at 8 p.m on Dec 10, 2020' FROM 'will.+');
```

	substring text	substring text
1	movie will start at 8 p.m on Dec 10, 2020	will start at 8 p.m on Dec 10, 2020

I can use the dot and plus to match everything after a specific word in the string

```
SELECT
substring ('The movie will start at 8 p.m on Dec 10, 2020' FROM '\w'),
substring ('The movie will start at 8 p.m on Dec 10, 2020' FROM '\w+');
```

	substring text	substring text
1	T	The

\w will get the word from the start of the string. If I want the whole word I use \w+

```
SELECT
substring ('The movie will start at 8 p.m on Dec 10, 2020' FROM '\w$'),
substring ('The movie will start at 8 p.m on Dec 10, 2020' FROM '\w+$');
```

	substring text	substring text
1	0	2020

\w\$ will get the word from the end of the string. If I want the whole word I use \w+\$

```
SELECT
substring ('The movie will start at 8 p.m on Dec 10, 2020' FROM '\d'),
substring ('The movie will start at 8 p.m on Dec 10, 2020' FROM '\d{2}'),
substring ('The movie will start at 8 p.m on Dec 10, 2020' FROM '\d{4}');
```

	substring text	substring text	substring text
1	8	10	2020

The `\d` notation will get the digits. `\d{m}` will get the number of digits i.e. `\d{2}` will get the first occurrence where it is two digits long, `\d{4}` will get the first occurrence where it is 4 digits long.

```
SELECT
substring ('The movie will start at 8 p.m on Dec 10, 2020' FROM '\d{1,2}'),
substring ('The movie will start at 10 p.m on Dec 10, 2020' FROM '\d{1,2} (a.m|p.m)'),
substring ('The movie will start at 10 p.m on Dec 10, 2020' FROM '\d{1,2} (?:a.m|p.m)');
```

	substring text	substring text	substring text
1	8	p.m	10 p.m

The `\d{m,n}` notation will get the first occurrence of a digit that is between 1 and 2 characters long. I can then add `(a.m|p.m)` whereby the between parenthesis is looking for a specific capture group of a.m. or p.m. the `?:` negates a capture group.

```
SELECT
substring ('The movie will start at 8 p.m on Dec 10, 2020' FROM '\d{4}'),
substring ('The movie will start at 10 p.m on Dec 10, 2020' FROM '(Nov|Dec)');
```

	substring text	substring text
1	2020	Dec

`\d{4}` will capture the first 4 digit number, `(Nov|Dec)` will capture either Nov or Dec.

```
SELECT
substring ('The movie will start at 8 p.m on Dec 10, 2020' FROM 'Dec \d{2}'),
substring ('The movie will start at 10 p.m on Dec 10, 2020' FROM 'Dec \d{2}, \d{4}');
```

	substring text	substring text
1	Dec 10	Dec 10, 2020

I can combine the above to pattern match the whole date, notice that where there is a comma in the string I include a comma in the REGEX also where there is a space in the string I include it in the Regex.

25.6 REGEXP_MATCHES Function

REGEXP_MATCHES() function matches a regular expression against a string and returns a set of matched text substrings.

REGEXP_MATCHES(source_string, pattern [, flags]);

Flags: g = Global search
I = match case-insensitively

```
SELECT  
REGEXP_MATCHES('Amazing #PostgreSQL', '#([A-Za-z0-9_])'),  
REGEXP_MATCHES('Amazing #PostgreSQL', '#([A-Za-z0-9_]+)');
```

	regexp_matches text[]	regexp_matches text[]
1	{P}	{PostgreSQL}

I can match anything after the hash using REGEX to find all capital, lower case and numbers. I use the plus to match multiple times.

```
SELECT  
REGEXP_MATCHES('Amazing #PostgreSQL #SQL', '#([A-Za-z0-9_]+)', 'g');
```

	regexp_matches text[]
1	{PostgreSQL}
2	{SQL}

If I use the Global flag it matches all instances. Note how the result set has two rows, i.e. two matches. Each is an array denoted by curly brackets.

```
SELECT REGEXP_MATCHES('XYZ', '^X(..)$', 'g');
```

	regexp_matches text[]
1	{X,YZ}

Group for the X then match everything at the end using the dot notation to match any character

```
SELECT REGEXP_MATCHES('My cat always jumps around', 'cat|dog', 'g');
```

	regexp_matches text[]
1	{cat}

The pipe for alternation is an either or which is Boolean but with REGEXP_MATCHES it returns the matching value, not a Boolean.

```
SELECT REGEXP_MATCHES('1111 2222-A 3333-B 4444-C', '-?[A-Z]', 'g');
```

I want to get the dash and character at the end of each chunk.

	regexp_matches text[]
1	{-A}
2	{-B}
3	{-C}

25.7 REGEXP_REPLACE Function

REGEXP_REPLACE() function to replace strings that match a regular expression.

REGEXP_REPLACE(source, pattern, replacement_string [, flags]);

Function returns a new string with the substrings, which match a regular expression pattern, replaced by a new substring.

```
SELECT  
REGEXP_REPLACE('Elliott Farmer', '', '\2 \1'),  
REGEXP_REPLACE('Elliott Farmer', '(.*) (.*)', '\2 \1');
```

	regexp_replace text	regexp_replace text
1	Elliott Farmer	Farmer Elliott

\2 matches the 2nd capturing group, \1 the first capturing group. So I can replace all characters from the 1st capturing group (.*?) then a space then the second capturing group (.*)

```
SELECT  
REGEXP_REPLACE('ABCD12345xyz', '[[[:digit:]]', '', 'g'),  
REGEXP_REPLACE('ABCD12345xyz', '[[[:alpha:]]', '', 'g');
```

	regexp_replace text	regexp_replace text
1	ABCDxyz	12345

Strip out the numeric characters from a string and strip out the alphabet characters from a string


```
SELECT
  REGEXP_REPLACE('2019-10-10', '\d{4}', '2020');
```

	regexp_replace text
1	2020-10-10

Replace the year with 2020

25.8 REGEXP_SPLIT_TO_TABLE Function

	regexp_split_to_table text
1	1
2	2
3	3
4	4

```
SELECT
  REGEXP_SPLIT_TO_TABLE('1,2,3,4', ',', '');
```

Splits delimited text into rows. The output is always text

```
SELECT
  REGEXP_SPLIT_TO_TABLE('Elliott,Ward,Farmer', ',', '');
```

	regexp_split_to_table text
1	Elliott
2	Ward
3	Farmer

25.9 REGEXP_SPLIT_TO_ARRAY Function

```
SELECT
  REGEXP_SPLIT_TO_ARRAY('Elliott,Ward,Farmer', ',', ' '),
  REGEXP_SPLIT_TO_ARRAY('Elliott Ward Farmer', ' ', ' '),
  ARRAY_LENGTH(REGEXP_SPLIT_TO_ARRAY('Elliott Ward Farmer', ' '), 1);
```

	regexp_split_to_array text[]	regexp_split_to_array text[]	array_length integer
1	{Elliott,Ward,Farmer}	{Elliott,Ward,Farmer}	3

Splits delimited text into arrays. The output is always text array {}.

26. Powerful Text Searches in PostgreSQL

[26.1 The Good ol' Text Search](#)

[26.2 Introducing tsvector](#)

[26.3 Using tsquery with Operators](#)

[26.4 Full Text Search Within a Table](#)

[26.5 Setup Presidents Speeches Data](#)

[26.6 Analysing Presidents Speeches Data](#)

[26.7 Ranking and Normalising Text results](#)

26.1 The Good ol' Text Search

```
SELECT * FROM movies WHERE movie_name LIKE '%star%';
```

The above query will return rows where column_name matches the pattern. But PostgreSQL is an RDBMS capable database capable of much more than simply storing and retrieving data.

What is full text search?

A full text search refers to a technique for searching a single computer-stored document or a collection in a full text database.

Distinguished from searches based on metadata or on parts of the original texts represented in databases (such as titles, abstracts, selected sections, or bibliographical references).

Problem:

In other words, imagine you have a set of text documents stored in a database. These documents are not just metadata items like an author name or a country of origin, but rather an abstract for an article, or full text articles themselves, and you want to find out if certain words are present or not in them.

For example, you may want to search if the nouns “dog” and “fox” are present so if they are in their singular form, you will find them with the LIKE keyword.

```
SELECT FROM table_name WHERE column_name LIKE '%dog%' AND column_name LIKE '%fox%';
```

The above query could return results that contain ‘dogville’ and foxtrot. Equally a search for ‘%query%’ would not find ‘queries’. Searching could include all the tenses of a verb such as ‘Jump’, ‘Jumped’, ‘jumping’ etc but this is not very practical.

Solution

A more effective approach to this problem is by getting a **semantic vector** for all the words contained in the document, that is a language specific representation of such words.

Then search the vector, not the full document.

26.2 Introducing tsvector

PostgreSQL comes with a powerful full text search engine that gives us more options when searching in a large amount of text.

Full Text Data types:

tsvector	Text to be searched and stored in optimized format
tsquery	Represents the search query terms and operators

tsvector

- Reduce text to a sorted list of 'lexemes', which are units of meaning of a language.
- Lexemes are words without the variations created by suffixes.
- E.G. tsvector will store words: washes, washed, and washing as 'wash'
- Removes 'stop words' like 'the', 'it' etc that usually don't play a role in search.

To_tsvector() function

To_tsvector for creating a list of tokens (the tsvector data type, where 'ts' stands for "text search")

```
SELECT  
to_tsvector('washes'),  
to_tsvector('washed'),  
to_tsvector('washing');
```

	to_tsvector tsvector	to_tsvector tsvector	to_tsvector tsvector
1	'wash':1	'wash':1	'wash':1

Data type: tsvector

Lexeme is 'wash'

```
SELECT to_tsvector('The quick brown fox jumped over  
the lazy dog');
```

	to_tsvector tsvector
1	'brown':3 'dog':9 'fox':4 'jump':5 'lazy':8 'quick':2

- Every token is a lexeme (unit of lexical meaning) with

The quick brown fox jumped over the lazy dog
1 2 3 4 5 6 7 8 9

- Pointers (the position in the document), and

'brown':3 'dog':9 'fox':4 'jump':5 'lazy':8 'quick':2

- Where words that carry little meaning such as articles (the) and conjunctions (and, or) are conveniently omitted.

- Orders the words alphabetically, and the number following each colon is its position in the original text.

```
SELECT to_tsvector('Hey wait, I am walking across  
the sitting room to sit with you');
```

	to_tsvector tsvector
1	'across':6 'hey':1 'room':9 'sit':8,11 'wait':2 'walk':5

Hey wait, I am walking across the sitting room to sit with you
1 2 3 4 5 6 7 8 9 10 11 12 13

26.3 Using tsquery with Operators

```
SELECT
to_tsvector('this is a test'),
to_tsvector('This is an amazing story'),
to_tsvector('postgresql tsvector text search is very powerful'),
to_tsvector('walking & sitting');
```

	to_tsvector tsvector	to_tsvector tsvector	to_tsvector tsvector	to_tsvector tsvector
1	'test':4	'amaz':4 'stori':5	'postgresq':1 'power':7 'search':4 'text':3 'tsvector':2	'sit':2 'walk':1

To_tsquery(text);

Accepts a list of words that will be checked against the normalised vector created with **to_tsvector()**

Operators

@@ Match operators

&& AND

| OR

! NOT

<> Search for adjacent words or words a certain distance apart

```
SELECT to_tsvector('This is a lamp')
@@ to_tsquery('lamp');
```

	?column? boolean
1	true

```
SELECT to_tsvector('This is a lamp')
@@ to_tsquery('light');
```

	?column? boolean
1	false

```
SELECT to_tsvector('The quick brown fox jumped over the lazy dog')  
@@ to_tsquery('fox');
```

	?column? boolean
1	true

```
SELECT to_tsvector('The quick brown fox jumped over the lazy dog')  
@@ to_tsquery('foxes');
```

	?column? boolean
1	true

```
SELECT to_tsvector('The quick brown fox jumped over the lazy dog')  
@@ to_tsquery('foxtrot');
```

	?column? boolean
1	false

The search is smart enough to find anything related to a fox but is not just searching '%fox%'. It is using the lexeme for fox. It also detects that 'foxtrot' has nothing to do with the animal fox and so does not match.

```
SELECT to_tsvector('The quick brown fox jumped over the lazy dog')  
@@ to_tsquery('fox & dog');
```

	?column? boolean
1	true

```
SELECT to_tsvector('The quick brown fox jumped over the lazy dog')  
@@ to_tsquery('fox | tiger');
```

	?column? boolean
1	true

```
SELECT to_tsvector('The quick brown fox jumped over the lazy dog')  
@@ to_tsquery('!tiger');
```

	?column? boolean
1	true

```
SELECT to_tsvector('The quick brown fox jumped over the lazy dog')  
@@ to_tsquery('!dog');
```

	?column? boolean
1	false

I can include search operators inside the tsquery such as or (|), not (!), and (&)

```
SELECT to_tsvector('The quick brown fox jumped over the lazy dog')
@@ to_tsquery('fox & dog | tiger & !king');
```

	?column? boolean
1	true

```
SELECT to_tsvector('The quick brown fox jumped over the lazy dog')
@@ to_tsquery('fox & (dog | tiger) & !king');
```

	?column? boolean
1	true

26.4 Full Text Search Within a Table

```
CREATE TABLE docs (
  doc_id SERIAL PRIMARY KEY,
  doc_text TEXT,
  doc_text_search TSVECTOR
);
```

```
SELECT *
FROM docs;
```

	doc_id [PK] integer	doc_text text	doc_text_search tsvector
1	1	The five boxing wizards jump quickly.	[null]
2	2	Pack my box with five dozen pepsi jugs.	[null]
3	3	How vexingly quick daft zebras jump!	[null]
4	4	Jackdaws love my big sphinx of quartz.	[null]
5	5	Sphinx of black quartz, judge my vow.	[null]
6	6	Bright vixens jump; dozy fowl quack.	[null]

```
INSERT INTO docs (doc_text) VALUES
('The five boxing wizards jump quickly.'),
('Pack my box with five dozen pepsi jugs.'),
('How vexingly quick daft zebras jump!'),
('Jackdaws love my big sphinx of quartz.'),
('Sphinx of black quartz, judge my vow.'),
('Bright vixens jump; dozy fowl quack.');
```

```
UPDATE docs
SET doc_text_search = to_tsvector(doc_text);
```

	doc_id [PK] integer	doc_text text	doc_text_search tsvector
1	1	The five boxing wizards jump quickly.	'box':3 'five':2 'jump':5 'quick':6 'wizard':4
2	2	Pack my box with five dozen pepsi jugs.	'box':3 'dozen':6 'five':5 'jug':8 'pack':1 'pepsi':7
3	3	How vexingly quick daft zebras jump!	'daft':4 'jump':6 'quick':3 'vex':2 'zebra':5
4	4	Jackdaws love my big sphinx of quartz.	'big':4 'jackdaw':1 'love':2 'quartz':7 'sphinx':5
5	5	Sphinx of black quartz, judge my vow.	'black':3 'judg':5 'quartz':4 'sphinx':1 'vow':7
6	6	Bright vixens jump; dozy fowl quack.	'bright':1 'dozi':4 'fowl':5 'jump':3 'quack':6 'vixen':2

```
SELECT * FROM docs;
```



```
SELECT
doc_id,
doc_text
FROM docs
WHERE doc_text_search @@ to_tsquery('jump');
```

	doc_id [PK] integer	doc_text text
1	1	The five boxing wizards jump quickly.
2	3	How vexingly quick daft zebras jump!
3	6	Bright vixens jump; dozy fowl quack.

```
SELECT
doc_id,
doc_text
FROM docs
WHERE doc_text_search @@ to_tsquery('jump & quick');
```

	doc_id [PK] integer	doc_text text
1	1	The five boxing wizards jump quickly.
2	3	How vexingly quick daft zebras jump!

```
SELECT
doc_id,
doc_text
FROM docs
WHERE doc_text_search @@ to_tsquery('jump <-> quick');
```

	doc_id [PK] integer	doc_text text
1	1	The five boxing wizards jump quickly.

<-> means that the words are side by side or next to each other.

```
SELECT
doc_id,
doc_text
FROM docs
WHERE doc_text_search @@ to_tsquery('sphinx <-> quartz');
```

	doc_id [PK] integer	doc_text text

There are no occurrences of 'sphinx' and 'quartz' lexemes occurring side by side.

```
SELECT
doc_id,
doc_text
FROM docs
WHERE doc_text_search @@ to_tsquery('sphinx <2> quartz');
```

	doc_id [PK] integer	doc_text text
1	4	Jackdaws love my big sphinx of quartz.

If I use '<2>' what I am saying is that the lexeme for 'sphinx' is two words apart from 'quartz'

	doc_id [PK] integer	doc_text text	doc_text_search tsvector
1	1	The five boxing wizards jump quickly.	'box':3 'five':2 'jump':5 'quick':6 'wizard':4
2	2	Pack my box with five dozen pepsi jugs.	'box':3 'dozen':6 'five':5 'jug':8 'pack':1 'pepsi':7
3	3	How vexingly quick daft zebras jump!	'daft':4 'jump':6 'quick':3 'vex':2 'zebra':5
4	4	Jackdaws love my big sphinx of quartz.	'big':4 'jackdaw':1 'love':2 'quartz':7 sphinx:5
5	5	Sphinx of black quartz, judge my vow.	'black':3 'judg':5 'quartz':4 sphinx:1 vow:7
6	6	Bright vixens jump; dozy fowl quack.	'bright':1 'dozi':4 'fowl':5 'jump':3 'quack':6 'vixen':2

What is going on under the hood is that it is matching the lexemes of 'sphinx' and 'quartz' then testing if they are two words apart, i.e. $7 - 5 = 2$ (**true**). Note how it has only matched on doc_id 4. if I compare doc_id 5 the lexemes of 'sphinx' and 'quartz' are $4 - 1 = 3$ (**false**) words apart.

This is what makes postgreSQL so fast at performing complex word matches in the tsquery.

```
SELECT
doc_id,
doc_text
FROM docs
WHERE doc_text_search @@ to_tsquery('sphinx <3> quartz');
```

	doc_id [PK] integer	doc_text text
1	5	Sphinx of black quartz, judge my vow.

```
SELECT
doc_id,
doc_text
FROM docs
WHERE doc_text_search @@ to_tsquery('quartz <3> sphinx');
```

	doc_id [PK] integer	doc_text text

	doc_id [PK] integer	doc_text text	doc_text_search tsvector
1	1	The five boxing wizards jump quickly.	'box':3 'five':2 'jump':5 'quick':6 'wizard':4
2	2	Pack my box with five dozen pepsi jugs.	'box':3 'dozen':6 'five':5 'jug':8 'pack':1 'pepsi':7
3	3	How vexingly quick daft zebras jump!	'daft':4 'jump':6 'quick':3 'vex':2 'zebra':5
4	4	Jackdaws love my big sphinx of quartz.	'big':4 'jackdaw':1 'love':2 'quartz':7 'sphinx':5
5	5	Sphinx of black quartz, judge my vow.	'black':3 'judg':5 'quartz':4 'sphinx':1 'vow':7
6	6	Bright vixens jump; dozy fowl quack.	'bright':1 'dozi':4 'fowl':5 'jump':3 'quack':6 'vixen':2

If I swap the order of the words around it does not match anything because **1 – 4 = -3 (false)** which is not the same as 3. equally, **7 – 5 = -2 (false)** which is not the same as 3.

So word1 <x> word2. word 2 has to be x words after word 1 for this operator to return a true.

26.5 Setup Presidents Speeches Data

- 1. Create table.
- 2. Download presidents speeches csv

C:\ path to \president_speeches.csv

northwind.sql	10/01/2023 22:40	SQL Source File	342 KB
president_speeches.csv	13/02/2025 14:52	Microsoft Excel C...	1,147 KB
products.sql	11/06/2023 22:36	SQL Source File	1 KB
purchases.sql		SQL Source File	2 KB
sample_data.zip		Compressed (zipp...	1,441 KB

- 3. Use the copy command to copy to contents of the CSV into the table

```
movies=# \copy docs_presidents (president, title, speach_date, speach_text) FROM 'C:/ path to /president_speeches.csv' WITH (FORMAT CSV, DELIMITER '|', HEADER OFF, QUOTE '@');
COPY 35
movies=#
```

- 4. verify

```
SELECT * FROM docs_presidents;
```

Showing rows: 1 to 35 Page No: 1 of 1					
	doc_id [PK] integer	president character varying (100)	title character varying (250)	speach_date date	speach_text text
1	1	Harry S. Truman	Message to the Congress on the State of the Union and on the Budget for 1946	1946-01-21	To the Congress of the United States:
2	2	Harry S. Truman	Annual Message to the Congress on the State of the Union	1947-01-06	Mr. President, Mr. Speaker, Members of the Congr
3	3	Harry S. Truman	Annual Message to the Congress on the State of the Union	1948-01-07	Mr. President, Mr. Speaker, and Members of the 80
4	4	Harry S. Truman	Annual Message to the Congress on the State of the Union	1949-01-05	Mr. President, Mr. Speaker, Members of the Congr

5. Convert speech text to tsvector, note that the first parameter of to_tsvector() is the language where if not specified it will default to English.

```
UPDATE docs_presidents
SET speech_text_search = to_tsvector('English', speech_text);
UPDATE 35
Query returned successfully in 311 msec.
```

6. Verify tsvector column

```
SELECT
doc_id
president
speech_text_search
FROM docs_presidents;
```

	doc_id [PK] integer	president character varying (100)	speech_text_search tsvector
1	1	Harry S. Truman	'-11':16383 '-17':12335 '-1945':704 '-39':12617 '...':4123 '000':989,5456,5467,6290,6303,7819,7825
2	2	Harry S. Truman	'-1948':4371 '000':2925,4952,5515,5588 '070':5514 '1':4208,5513 '15':2593 '1926':2961 '1941':26
3	3	Harry S. Truman	'1':2360,4112,4743 '10':469,480,483,533,699,2085,2093,2144,2216,4436,4447 '12':1010 '12.5':47
4	4	Harry S. Truman	'1':2431 '16':73,3217 '1929':476 '1945':3203 '1946':963 '1947':1445 '2':1473 '20':484 '25':2134 '25
5	5	Harry S. Truman	'1930':1604 '1948':4316 '1949':3413 '1950':362 '2':1443 '2000':2570 '20th':368,4866,5155 '255':6
6	6	Harry S. Truman	'-17':2943,2992,2999 '-36':2959,2982,3007 '000':2860,2874,3002 '1/2':2580 '1931':2196 '1935':22
7	7	Harry S. Truman	'120':1042 '13':2484 '15':1036 '1948':2462 '1951':275 403 469 562 639 1135 1250 '1952':307 349

7. Create special index for tsvector. By default PostgreSQL uses a btree index but GIN index is faster for text based data.

```
CREATE INDEX idx_gin_docs_presidents_speech_text_search ON docs_presidents USING
GIN(speech_text_search);
CREATE INDEX
Query returned successfully in 62 msec.
```

26.6 Analysing Presidents Speeches

```
SELECT
president,
title,
speech_date
FROM docs_presidents
WHERE speech_text_search @@ to_tsquery('world');
```

All the presidents speeches contain the lexeme 'world'

Showing rows: 1 to 35  Page No: 1			
	president character varying (100) 	title character varying (250) 	speech_date date 
1	Harry S. Truman	Message to the Congress on the State of the Union and on the Budget for 1946	1946-01-21
2	Harry S. Truman	Annual Message to the Congress on the State of the Union	1947-01-06
3	Harry S. Truman	Annual Message to the Congress on the State of the Union	1948-01-07

```
SELECT
president,
title,
speech_date
FROM docs_presidents
WHERE speech_text_search @@ to_tsquery('humanity');
```

29 speeches contain a reference to the lexeme 'humanity'

```
SELECT
president,
title,
speech_date
FROM docs_presidents
WHERE speech_text_search @@ to_tsquery('war');
```

34 speeches contain a reference to the lexeme 'war'

```
SELECT
president,
title,
speach_date
FROM docs_presidents
WHERE speach_text_search @@ to_tsquery('war & peace');
```

34 speeches contain a reference to the lexeme 'war' and 'peace' anywhere in the whole text document.

```
SELECT
president,
title,
speach_date
FROM docs_presidents
WHERE speach_text_search @@ to_tsquery('god & bless');
```

8 speeches contain a reference to the lexeme 'god' and 'bless' anywhere in the whole text document.

```
SELECT
president,
title,
speach_date
FROM docs_presidents
WHERE speach_text_search @@ to_tsquery('military <-> defense');
```

4 speeches contain a reference to the directly adjacent lexemes of 'military', and 'defense'

	president character varying (100) 	title character varying (250) 	speach_date date 
1	Dwight D. Eisenhower	Annual Message to the Congress on the State of the Union	1956-01-05
2	Dwight D. Eisenhower	Annual Message to the Congress on the State of the Union	1958-01-09
3	Dwight D. Eisenhower	Annual Message to the Congress on the State of the Union	1959-01-09
4	Richard M. Nixon	Address on the State of the Union Delivered Before a Joint Session of the Congress	1972-01-20

```
SELECT
president,
title,
speech_date,
ts_headline(speech_text, to_tsquery('military <-> defense'));
FROM docs_presidents
WHERE speech_text_search @@ to_tsquery('military <-> defense');
```

	president character varying (100)	title character varying (250)	speech_date date	ts_headline text
1	Dwight D. Eisenhower	Annual Message to the C...	1956-01-05	military defenses are designed to protect.
2	Dwight D. Eisenhower	Annual Message to the C...	1958-01-09	military defense efforts, but likewise in our foreign policy, through such activities a
3	Dwight D. Eisenhower	Annual Message to the C...	1959-01-09	military defense of national life and territory. It is the preservation of a way of life
4	Richard M. Nixon	Address on the State of t...	1972-01-20	military defenses are not the enemy of peace; they are the guardians of peace.

The `ts_headline()` function can be used to find the specified lexemes and output the context of where these words appear within the text. Note how it has given the lexemes a bold HTML tag as well as the surrounding context text. I can change this by specifying a start select and end select.

```
SELECT
president,
title,
speech_date,
ts_headline(speech_text, to_tsquery('military <-> defense'), 'StartSel=<,StopSel=>')
FROM docs_presidents
WHERE speech_text_search @@ to_tsquery('military <-> defense');
```

	president character varying (100)	title character varying (250)	speech_date date	ts_headline text
1	Dwight D. Eisenhower	Annual Message to the Congress on t...	1956-01-05	<military> <defenses> are designed to protect.
2	Dwight D. Eisenhower	Annual Message to the Congress on t...	1958-01-09	<military> <defense> efforts, but likewise in our foreign policy, through s
3	Dwight D. Eisenhower	Annual Message to the Congress on t...	1959-01-09	<military> <defense> of national life and territory. It is the preservation o
4	Richard M. Nixon	Address on the State of the Union Deli...	1972-01-20	<military> <defenses> are not the enemy of peace; they are the guardian

26.7 Ranking and Normalising Text results

```
SELECT
president,
title,
speech_date,
ts_rank(speech_text_search, to_tsquery('military <-> defense')) AS score
FROM docs_presidents
WHERE speech_text_search @@ to_tsquery('military <-> defense')
ORDER BY score DESC;
```

	president character varying (100) 🔒	title character varying (250) 🔒	speech_date date 🔒	score real 🔒
1	Dwight D. Eisenhower	Annual Message to the Congress on the State of the Union	1958-01-09	0.5864567
2	Dwight D. Eisenhower	Annual Message to the Congress on the State of the Union	1959-01-09	0.19083127
3	Richard M. Nixon	Address on the State of the Union Delivered Before a Joint Session of the Congress	1972-01-20	0.18696731
4	Dwight D. Eisenhower	Annual Message to the Congress on the State of the Union	1956-01-05	0.09920868

Ranking attempts to measure how relevant documents are to a particular query, so that when there are many matches the most relevant ones can be shown first. PostgreSQL provides two predefined ranking functions, which take into account lexical, proximity, and structural information; that is, they consider **how often the query terms appear in the document**, how close together the terms are in the document

<https://www.postgresql.org/docs/current/textsearch-controls.html#:~:text=Ranking%20attempts%20to%20measure,are%20in%20the%20document>

Typically ranking would give a higher rank to text that contains more occurrences of the searched lexemes. A speech that has 'military defense' many times is ranking higher than a speech that has fewer occurrences but this can be normalised by adding a normalisation parameter, i.e. 2

```
SELECT
president,
title,
speech_date,
ts_rank(speech_text_search, to_tsquery('military <-> defense'),2) AS score
FROM docs_presidents
WHERE speech_text_search @@ to_tsquery('military <-> defense')
ORDER BY score DESC;
```

	president character varying (100) 	title character varying (250) 	speech_date date 	score real 
1	Dwight D. Eisenhower	Annual Message to the Congress on the State of the Union	1958-01-09	0.00021752846
2	Richard M. Nixon	Address on the State of the Union Delivered Before a Joint Session of the Congress	1972-01-20	9.5635456e-05
3	Dwight D. Eisenhower	Annual Message to the Congress on the State of the Union	1959-01-09	7.067825e-05
4	Dwight D. Eisenhower	Annual Message to the Congress on the State of the Union	1956-01-05	2.1534335e-05

27. Table Partitions

[27.1 What is a Partition?](#)

[27.2 When a Partition can be Used](#)

[27.3 Table Inheritance](#)

[27.4 Partition Types](#)

[27.5 Partition by Range](#)

[27.6 Partition by List](#)

[27.7 Partition by Hash](#)

[27.8 DEFAULT Partition](#)

[27.9 Multi-Level Partitioning](#)

[27.10 Partition Maintenance](#)

[27.11 ALTERing the Partition Bounds](#)

[27.12 Partition Indexes](#)

[27.13 Partition Pruning](#)

[27.14 Determining a field for Partition Over](#)

[27.15 Sizing the Partition](#)

[27.16 Partitioning Advantages](#)

[27.17 Common Partitioning Mistakes](#)

27.1 What is a Partition?

1. Partitioning is splitting one table into:

- Multiple smaller
- Logically divided, and
- More manageable pieces tables

2. Partitioning leads to a huge performance boost for very large tables that could for example contain millions of rows of products.

```
SELECT * FROM products;  
SELECT * FROM products WHERE product_name LIKE '%A';
```

PostgreSQL is able to exclude partitions that, for sure, won't be affected by the data we are reading or writing. Read write only what is needed!

3. Partitioned tables can improve query performance by allowing the database query optimizer to scan only the data needed to satisfy a given query instead of scanning the whole contents of a large table.

27.2 When a Partition can be Used

1. When we have a large table or Is the table large enough?

- Large fact tables are good candidates for table partitioning. If you have millions or billions or records in a table
- Don't think about partitioning before reaching several million records, as there would be no gain in performance.
- Also, for smaller tables with only a few thousand rows or less (That is not what we want partition tables to be.) The administrative overhead of maintaining the partitions will outweigh any performance benefits you might see.

2. Your table qualify to be able to be logically divided into smaller chunks

- E.g. server logs, where you can split them by 'date' having all the records related to the same day in the same partition. Bulk operations like deleting old logs would be as simple as dropping a partition.

3. Experiencing unsatisfactory performance/Slow queries?

- Although always check the query plans, respective indexes to solve first
- Test with a partition table to see if the query execution times increased after the partition or not.

4. To avoid scanning the whole table, and have fast query processing

- Review indexes on a large table i.e. primary key, Foreign keys, and composite indexes before partitioning

5. In a large table, some columns are frequently occurring in the WHERE clauses.

- Examine the WHERE clauses of your query workload and look for table columns that are consistently used to access data.

```
SELECT * FROM large_table WHERE column <conditions>;
```

- Partition based on most frequent query condition. E.g.
 - Query looks by dates Partition by dates e.g. weekly, monthly tables etc
 - Query looks by region Partition by each region type e.g. city, country etc

6. Table can't fit into memory

- Strongly advise not to reach this limit, usually when a table hits the size of some gigabytes it is time to partition.
- Always try to achieve the efficient use of memory

7. Business requirements for maintaining historical data.

- Your data warehouse needs to maintain a window of historical data
- Past data is important but may not be needed for a query
- For example your data warehouse may require that you keep data for the past 12 months. If the data is partitioned by month, you can easily archive the oldest monthly partition from the warehouse and load current data into the most recent month partition.

8. If your data can be divided into equal parts based on some good criteria.

- When partitioning, try to divide your data as evenly as possible.
- Partitions containing a relatively equal number of records will boost query execution performance

9. Do not create more partitions than are needed

- Having too many partitions means an overhead on management maintenance tasks such as:
- vaccuming
- Recovering segments
- Expanding the cluster
- Checking disk usage
- And others.

10. Queries that scan every partition run slower than if the table were not partitioned correctly

11. Be very careful with multi-level partitioning because the number of partition files can grow very quickly.

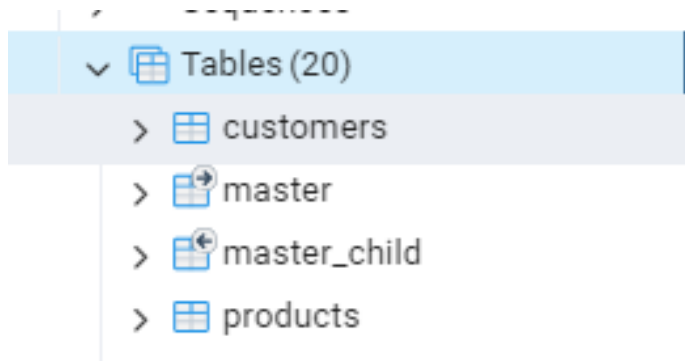
- For example, if a table is partitioned by both day and city, and there are 1,000 days of data and 1,000 cities the total number of partitions is 1 million.
- Also column-orientated tables will store each column in a physical table, so if this table has 100 columns, the system would be required to manage 1 million files for the table.
- Keep it simple Master level → level 1 → level 2. don't go more than 2 levels unless absolutely necessary.

It is not necessary that in every use case, we can find the field on which we want to split a large table into partition tables so look at the data → view most frequent queries → create test partition → improve bit by bit.

27.3 Table Inheritance

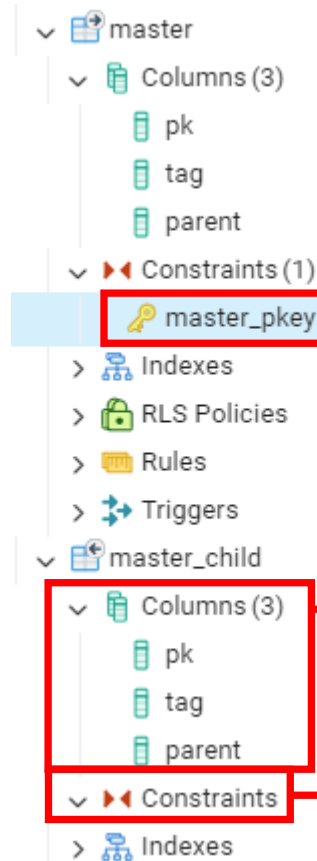
1. Inheritance is an object-orientated concept when a subclass inherits the properties of it's parent class.
2. In PostgreSQL, a table inherits from it's parent table
3. An example: master → master_child

```
CREATE TABLE master (  
  pk INTEGER PRIMARY KEY,  
  tag TEXT,  
  parent INTEGER  
);
```



Notice how the 'master' and 'master_child' table have a plus symbol on the table icon.

```
CREATE TABLE master_child() INHERITS (master);
```



The master_child table inherits all the properties of the master table.

The master table has a constraint on the primary key.

The master_child inherits column_names and properties from the master table

The master_child table does not have any constraints

master_child

General

Columns

Advanced

Constraints

Partitions

Parameters

Security

SQL

Inherited from table(s)

public.master x

Columns

+

	Name	Data type	Length/Precision	Scale	Not NULL?	Primary key?	Default
	pk	integer					
	tag	text					
	parent	integer					

Note that the pk column of the master_child table is not the primary key but it needs to be.

```
ALTER TABLE master_child
ADD CONSTRAINT master_pk PRIMARY KEY (pk);
```

```
SELECT * FROM master;
```

pk [PK] integer	tag text	parent integer

```
SELECT * FROM master_child;
```

pk [PK] integer	tag text	parent integer

```
learning-# \d master
Table "public.master"
Column | Type   | Collation | Nullable | Default
-----+-----+-----+-----+-----
pk      | integer |           | not null |
tag     | text    |           |          |
parent  | integer |           |          |
Indexes:
    "master_pkey" PRIMARY KEY, btree (pk)
Number of child tables: 1 (Use \d+ to list them.)
```

From the terminal I can describe the table and see that it has one child table

```
learning-# \d+ master
```

Table "public.master"								
Column	Type	Collation	Nullable	Default	Storage	Compression	Stats target	Description
pk	integer		not null		plain			
tag	text				extended			
parent	integer				plain			

Indexes:

"master_pkey" PRIMARY KEY, btree (pk)

Child tables: master_child

Access method: heap

```
learning-# \d+ master_child
```

Table "public.master_child"								
Column	Type	Collation	Nullable	Default	Storage	Compression	Stats target	Description
pk	integer		not null		plain			
tag	text				extended			
parent	integer				plain			

Indexes:

"master_pk" PRIMARY KEY, btree (pk)

Inherits: master

Access method: heap

```
INSERT INTO master (pk, tag, parent) VALUES (1,'pen',0);
INSERT INTO master_child (pk, tag, parent) VALUES (2,'pencil',0);
```

```
SELECT * FROM master;
```

	pk [PK] integer	tag text	parent integer
1	1	pen	0
2	2	pencil	0

```
SELECT * FROM master_child;
```

	pk [PK] integer	tag text	parent integer
1	2	pencil	0

Rows inserted into master_child table are shown in the master table. The master table contains all the records.

```
SELECT * FROM ONLY master;
```

	pk [PK] integer	tag text	parent integer
1	1	pen	0

```
SELECT * FROM ONLY master_child;
```

	pk [PK] integer	tag text	parent integer
1	2	pencil	0

If I use the ONLY keyword then I can see only the rows from the actual table, excluding inherited rows.

```
UPDATE master  
SET tag = 'monitor'  
WHERE pk = 2;
```

```
SELECT * FROM master;
```

	pk [PK] integer	tag text	parent integer
1	1	pen	0
2	2	monitor	0

```
SELECT * FROM master_child;
```

	pk [PK] integer	tag text	parent integer
1	2	monitor	0

If I update a row in the master table then the change is propagated to the child table.

```
DELETE FROM master WHERE pk = 2;
```

```
SELECT * FROM master;
```

	pk [PK] integer	tag text	parent integer
1	1	pen	0

```
SELECT * FROM master_child;
```

	pk [PK] integer	tag text	parent integer

Deleting a row in the master table will also cause the corresponding row of the child table to be deleted.

```
DROP table master;  
ERROR: table master_child depends on table mastercannot drop table master because  
other objects depend on it  
ERROR: cannot drop table master because other objects depend on it  
SQL state: 2BP01  
Detail: table master_child depends on table master  
Hint: Use DROP ... CASCADE to drop the dependent objects too.
```

Dropping a master table that has child table dependencies will produce an error.

```
DROP TABLE master CASCADE;  
NOTICE: drop cascades to table master_child  
DROP TABLE Query returned successfully in 47 msec.
```

Adding CASCADE to the drop will drop the master table and the child_master table. Note that this is a non reversible command and if **used in a production environment may cause unplanned data loss.**

In PostgreSQL 12 or higher, inheritance propagates the operations performed on the parent table to the child table.

The parent child table relationship is what is used by partitioning.

27.4 Partition Types

1. Range The table is partitioned into ranges defined by a key column or set of columns, with no overlap between the ranges of values assigned to different partitions.

Quarters: Oct-2019 to Dec-2019, Jan-2020 to Mar2020, Apr-2020 to Jun-2020, Jul-2020 to Sep-2020

Years: tbl_2020 tbl_2019

1. List The table is partitioned by explicitly listing which key values appear in each partition.

Countries: country_us, country_mx, country_ca, country_uk

1. Hash The table is partitioned by specifying a modulus and a remainder for each partition. Each partition will hold the rows for which the hash value of the partition key divided by the specified modulus will produce the specified remainder.

27.5 Partition by Range

The table is partitioned into “ranges” defined by a key column or set of columns, with NO OVERLAP between the ranges of values assigned to different partitions.

Useful when working with dates e.g. Daily / Weekly / Monthly / Yearly / partitions

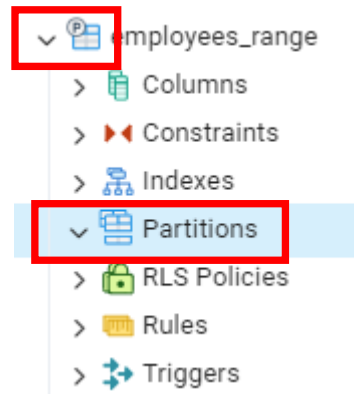
Consider partitioning by the most granular level.

For example, for a table partitioned by date, you can partition by:

Day – 365 daily partitions // Month – 12 partitions // Year – then sub partition by month
// Month – then sub partition by day

A multi-level design can reduce query planning time, but a flat partition design runs faster.

```
CREATE TABLE employees_range (  
  id BIGSERIAL,  
  birth_date DATE NOT NULL,  
  country_code VARCHAR(2) NOT NULL  
) PARTITION BY RANGE (birth_date);
```



Note that the table has a p symbol in the icon as well as an extra menu called partition.

Plan the partition. In this case we will partition by year.

FROM value1 TO value2. Note for dates the value 2 is important!

i.e. 2000-01-01 TO 2001-01-01. you would think that it would need to be to 2000-12-31 but no. Increment the year by 1 if partitioning by year.

So we want a partition table for yearly partitions

Tablename employees_range_y2000 keep the naming convention logical i.e. master table name then the partition name.

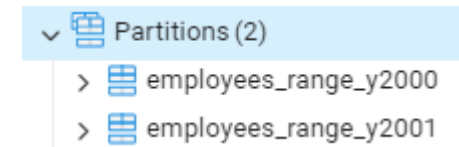
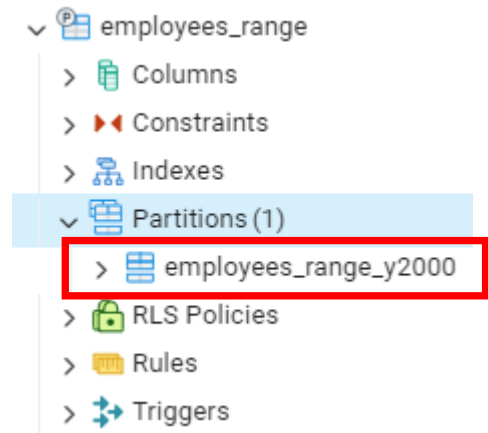
```
CREATE TABLE employees_range_y2000 PARTITION OF employees_range  
FOR VALUES FROM ('2000-01-01') TO ('2001-01-01')
```

I create a new table naming it according to the partition convention then I use PARTITION OF keyword and specify the master table. I define the values that I want in the partition table using a range.

Note that now under the partition icon I have a partition table.

```
CREATE TABLE employees_range_y2001 PARTITION OF employees_range  
FOR VALUES FROM ('2001-01-01') TO ('2002-01-01')
```

```
INSERT INTO employees_range (birth_date, country_code) VALUES  
( '2000-01-01', 'US'),  
( '2000-01-02', 'US'),  
( '2000-12-31', 'US'),  
( '2001-01-01', 'US');
```



Insert into the master table
not the child partition table.

```
SELECT * FROM employees_range;
```

	id bigint	birth_date date	country_code character varying (2)
1	1	2000-01-01	US
2	2	2000-01-02	US
3	3	2000-12-31	US
4	4	2001-01-01	US

All rows are in the master table. Also the year 2000 rows have been copied to the y2000 partition. The y2001 portion has also received a row.

```
SELECT * FROM employees_range_y2000;
```

	id bigint	birth_date date	country_code character varying (2)
1	1	2000-01-01	US
2	2	2000-01-02	US
3	3	2000-12-31	US

```
SELECT * FROM ONLY employees_range;
```

	id bigint	birth_date date	country_code character varying (2)

Now if I use the ONLY keyword, I actually see that there are no rows stored in the master table. All the data is actually in the partition tables. NO DATA is stored in the MASTER TABLE.

```
UPDATE employees_range
SET birth_date = '2001-10-10'
WHERE id = 1;
```

Any updates are performed on the MASTER table, not the partition table. The master table will propagate the changes.

```
SELECT * FROM employees_range;
```

	id bigint	birth_date date	country_code character varying (2)
1	2	2000-01-02	US
2	3	2000-12-31	US
3	4	2001-01-01	US
4	1	2001-10-10	US

```
SELECT * FROM ONLY employees_range_y2000;
```

	id bigint	birth_date date	country_code character varying (2)
1	2	2000-01-02	US
2	3	2000-12-31	US

```
SELECT * FROM ONLY employees_range_y2001;
```

	id bigint	birth_date date	country_code character varying (2)
1	4	2001-01-01	US
2	1	2001-10-10	US


```
DELETE FROM employees_range  
WHERE id = 1;
```

```
SELECT * FROM employees_range;
```

	id bigint 	birth_date date 	country_code character varying (2) 
1	2	2000-01-02	US
2	3	2000-12-31	US
3	4	2001-01-01	US

Any Deletes are performed on the MASTER table, not the partition table. The master table will propagate the changes.

```
SELECT * FROM ONLY employees_range_y2000;
```

	id bigint 	birth_date date 	country_code character varying (2) 
1	2	2000-01-02	US
2	3	2000-12-31	US

```
SELECT * FROM ONLY employees_range_y2001;
```

	id bigint 	birth_date date 	country_code character varying (2) 
1	4	2001-01-01	US



Query Query History

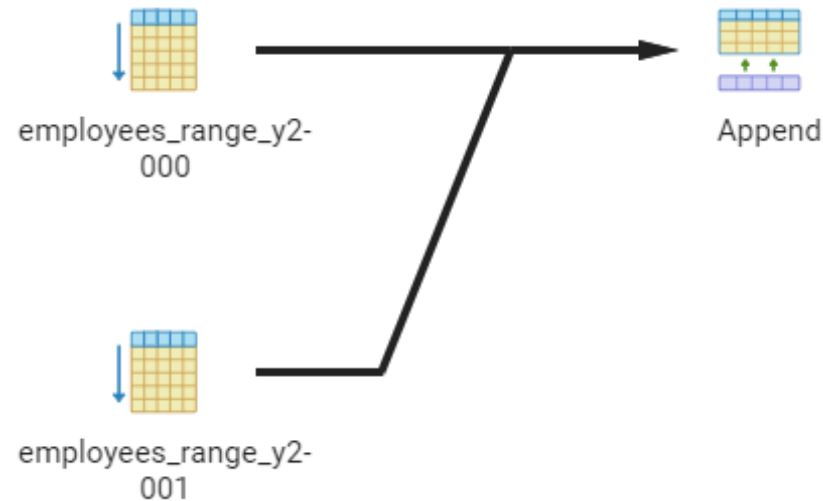
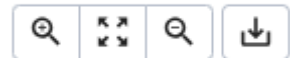
```
1 SELECT * FROM employees_range;
```

2

3

Data Output Messages Explain x Notifications

Graphical Analysis Statistics



If I run a query with a where clause to find by a particular date then the explain tells me the partition table that it is taking data from. This is the point of partitioning – reduce the number of rows it has to look in to find the data

Explain Analyze

Shift F7

If I run the query from PgAdmin4 tool and click the **explain button** it will show a diagram showing that it is taking all the data from 'employees_range_y2000' partition table and 'employees_range_y2001' partition and appending it to the 'employees_range' table for the results of this query.

Query Query History

```
1 SELECT * FROM employees_range
2 WHERE birth_date = '2000-01-02';
3
```

Data Output Messages Explain x Notifications

Graphical Analysis Statistics

employees_range_y2-000

27.6 Partition by List

The table is partitioned by listing which key values appear in each partition.

A list partitioned table can use any data type column that allows equality comparisons as its partition key column.

Can also have a multi-column (composite) partition key

Ideal for conditions when you know the values of a partition key i.e. country codes (iso country), month_names (Jan. Feb, Mar..) etc...

We create a partition by LIST when we want to create a partition on a list of values. E.g. if we have a master table called “employee_list” with country codes of US, UK, JP etc

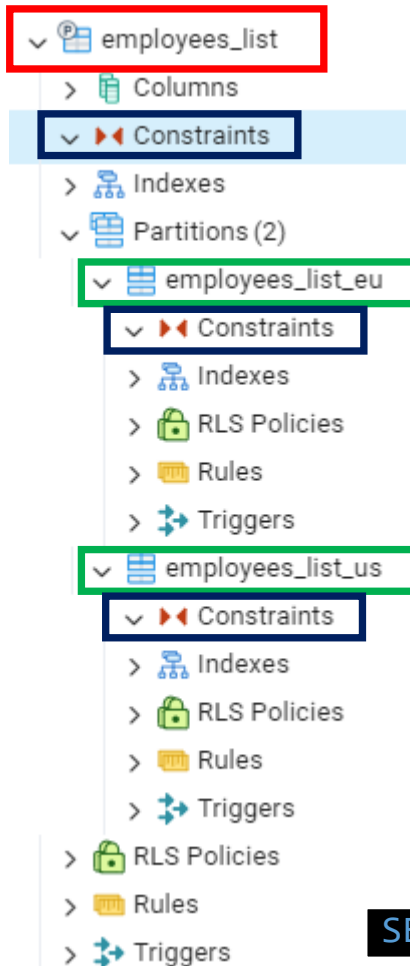
```
CREATE TABLE employees_list (  
  id BIGSERIAL,  
  birth_date DATE NOT NULL,  
  country_code VARCHAR(2) NOT NULL  
) PARTITION BY LIST (country_code);
```

Create the master table specifying partition by list with the column to partition on.

```
CREATE TABLE employees_list_us PARTITION OF employees_list  
FOR VALUES IN ('US');
```

```
CREATE TABLE employees_list_eu PARTITION OF employees_list  
FOR VALUES IN ('UK', 'DE', 'IT', 'FR', 'ES');
```

Create a partition table. Note that by using IN I can supply a list of values that I would like to be in a partition table.



I can see that the **master table** is created and has a p (partitioned) symbol on the icon, as well as the **two partition tables**. However there is **no PK constraint in the master of partition tables**.

```
INSERT INTO employees_list (birth_date, country_code) VALUES
('2000-01-01', 'US'),
('2000-01-02', 'US'),
('2000-12-31', 'UK'),
('2001-01-01', 'DE');
```

```
SELECT * FROM employees_list;
```

	id bigint	birth_date date	country_code character varying (2)
1	3	2000-12-31	UK
2	4	2001-01-01	DE
3	1	2000-01-01	US
4	2	2000-01-02	US

```
SELECT * FROM ONLY employees_list;
```

	id bigint	birth_date date	country_code character varying (2)

```
SELECT * FROM employees_list_us;
```

	id bigint	birth_date date	country_code character varying (2)
1	1	2000-01-01	US
2	2	2000-01-02	US

```
SELECT * FROM employees_list_eu;
```

	id bigint	birth_date date	country_code character varying (2)
1	3	2000-12-31	UK
2	4	2001-01-01	DE

As with a range list the master table does not contain any rows but is made up of data taken from the partition tables. Data is automatically inserted into the partition data based on the criteria specified I.e. country code. The **master table is in fact an aggregation of the partition tables**.

```
UPDATE employees_list
SET country_code = 'US'
WHERE id = 4;
UPDATE 1
Query returned successfully in 71 msec.
```

If I update a row by changing the country code...

```
SELECT * FROM employees_list_us;
```

	id bigint	birth_date date	country_code character varying (2)
1	1	2000-01-01	US
2	2	2000-01-02	US
3	4	2001-01-01	US

I can see that the record is moved to the new partition table.

```
SELECT * FROM employees_list;
```

	id bigint	birth_date date	country_code character varying (2)
1	3	2000-12-31	UK
2	1	2000-01-01	US
3	2	2000-01-02	US
4	4	2001-01-01	US

```
SELECT * FROM employees_list_eu;
```

	id bigint	birth_date date	country_code character varying (2)
1	3	2000-12-31	UK

ALWAYS MAKE CHANGES TO THE MASTER TABLE, NEVER ON A PARTITION TABLE

```
DELETE FROM employees_list
WHERE id = 2;
DELETE 1
Query returned successfully in 51 msec.
```

If I delete a row...

```
SELECT * FROM employees_list_us;
```

	id bigint	birth_date date	country_code character varying (2)
1	1	2000-01-01	US
2	4	2001-01-01	US

The row is removed from the relevant partition table

```
SELECT * FROM employees_list;
```

	id bigint	birth_date date	country_code character varying (2)
1	3	2000-12-31	UK
2	1	2000-01-01	US
3	4	2001-01-01	US

And does not show in the master table aggregation.

The screenshot shows the PgAdmin4 interface. The query editor contains the following SQL query:

```
1 SELECT * FROM employees_list;
```

The 'Explain' button is highlighted with a tooltip that says 'Explain Analyze' and 'Shift F7'. Below the query editor, the 'Data Output' tab is selected, and the 'Graphical' sub-tab is active. The graphical view shows a diagram with two table icons labeled 'employees_list_eu' and 'employees_list_us'. Arrows from both tables point to a single 'Append' icon, indicating that the query is retrieving data from all partitions.

If I run the query from PgAdmin4 tool and click the **explain button** it will show a diagram showing that it is taking all the data all partitions because a * query is all rows. This is not great improvement to a non partitioned table.

If I include where clauses and specify a value that is based on a partition then now the query is only looking in the relevant partition table.

The screenshot shows the PgAdmin4 interface with a more specific query:

```
1 SELECT * FROM employees_list
2 WHERE country_code = 'US';
```

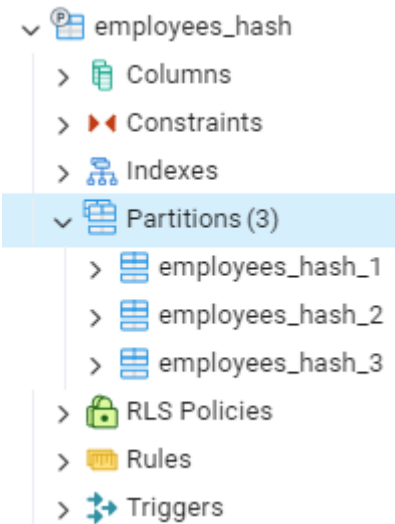
The 'Explain' button is highlighted with a tooltip that says 'Explain Analyze' and 'Shift F7'. Below the query editor, the 'Data Output' tab is selected, and the 'Graphical' sub-tab is active. The graphical view shows a diagram with a single table icon labeled 'employees_list_us', indicating that the query is only looking in the relevant partition.

27.7 Partition by Hash

The table is partitioned by a modulus and a remainder for each partition.

This type of partition is useful when we cannot logically divide our data by an existing column.

Use this type if you want to reduce the table size by spreading rows into many smaller partitions.



```
CREATE TABLE employees_hash (  
  id BIGSERIAL,  
  birth_date DATE NOT NULL,  
  country_code VARCHAR(2) NOT NULL  
 ) PARTITION BY HASH (id);
```

Create the master table specifying partition by hash with the column to partition on, in this case id.

```
CREATE TABLE employees_hash_1 PARTITION OF employees_hash  
FOR VALUES WITH (MODULUS 3, REMAINDER 0);
```

```
CREATE TABLE employees_hash_2 PARTITION OF employees_hash  
FOR VALUES WITH (MODULUS 3, REMAINDER 1);
```

```
CREATE TABLE employees_hash_3 PARTITION OF employees_hash  
FOR VALUES WITH (MODULUS 3, REMAINDER 2);
```

The partitions are created with a **range of values**. The range is specified using modulus and remainder.

```
INSERT INTO employees_hash (birth_date, country_code) VALUES  
( '2000-01-01', 'US' ),  
( '2000-01-02', 'US' ),  
( '2000-12-31', 'UK' );
```

```
SELECT * FROM employees_hash;
```

	id bigint	birth_date date	country_code character varying (2)
1	2	2000-01-01	US
2	4	2000-12-31	UK
3	3	2000-01-02	US

No records are stored in the master table. The rows are automatically inserted into the respective partition.

```
SELECT * FROM ONLY employees_hash;
```

	id bigint	birth_date date	country_code character varying (2)
1	2	2000-01-01	US
2	4	2000-12-31	UK
3	3	2000-01-02	US

```
UPDATE employees_hash  
SET country_code = 'US'  
WHERE id = 3;  
UPDATE 1  
Query returned successfully in 56 msec.
```

```
SELECT * FROM employees_hash;
```

	id bigint	birth_date date	country_code character varying (2)
1	2	2000-01-02	US
2	3	2000-12-31	US
3	1	2000-01-01	US

```
SELECT * FROM employees_hash_1;
```

	id bigint	birth_date date	country_code character varying (2)
1	2	2000-01-02	US

```
SELECT * FROM employees_hash_2;
```

	id bigint	birth_date date	country_code character varying (2)
1	3	2000-12-31	UK

```
SELECT * FROM employees_hash_3;
```

	id bigint	birth_date date	country_code character varying (2)
1	1	2000-01-01	US

Here we saw an insert statement on the master table 'employees_hash'. However, based on the value of the 'id' column, each row is added to the respective partition table and total rows of master table are the aggregation of all the partition tables.

```
SELECT * FROM employees_hash_2;
```

	id bigint	birth_date date	country_code character varying (2)
1	3	2000-12-31	US

Updating a row makes changes to that row in the relevant partition table.


```
DELETE FROM employees_hash
WHERE id = 2;
DELETE 1
Query returned successfully in 49 msec.
```

```
SELECT * FROM employees_hash_1;
```

	id bigint	birth_date date	country_code character varying (2)
1	3	2000-12-31	US

```
SELECT * FROM employees_hash_2;
```

	id bigint	birth_date date	country_code character varying (2)
1	3	2000-12-31	US

```
SELECT * FROM employees_hash;
```

	id bigint	birth_date date	country_code character varying (2)
1	3	2000-12-31	US
2	1	2000-01-01	US

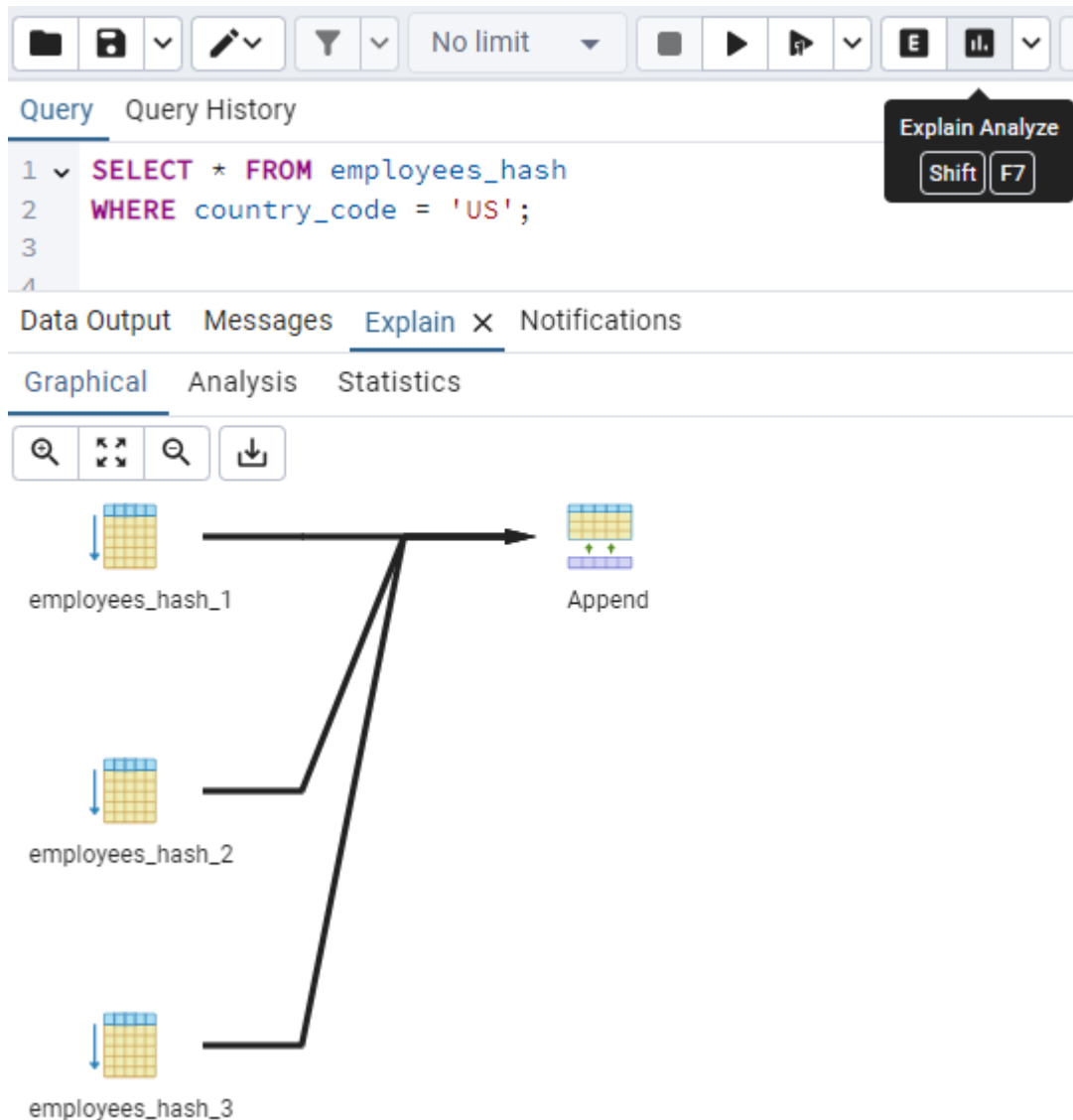
```
SELECT * FROM employees_hash_3;
```

	id bigint	birth_date date	country_code character varying (2)
1	1	2000-01-01	US

The delete and update operation only works on the corresponding partition table.

The screenshot shows the PgAdmin4 interface. At the top, there's a toolbar with icons for file operations, query execution, and explain/analyze. Below the toolbar, the 'Query' tab is active, showing a SQL query: `SELECT * FROM employees_hash WHERE id = 3;`. To the right of the query editor, there's a tooltip for the 'Explain Analyze' button, which is labeled 'Shift F7'. Below the query editor, there are tabs for 'Data Output', 'Messages', 'Explain', and 'Notifications'. The 'Explain' tab is selected, showing a graphical representation of the query plan. The graphical plan shows a table icon labeled 'employees_hash_2' with a blue arrow pointing to it, indicating the data source for the query.

If I run the query from PgAdmin4 tool and click the **explain button** it will show a diagram that explains where it is looking for the data. In this query I want to find a row based on an Id value which is the column on which I am partitioning the data so it knows it only has to look in the relevant partition to find that data.



If I run the query from PgAdmin4 tool and click the **explain button** it will show a diagram that explains where it is looking for the data. In this query I want to find a row based on an 'country_code' value which is NOT the column on which I am partitioning the data so it has scan all partitions to find the row.

This is why it is important to partition on the column that is most frequently filtered for in queries. When a query has to look in all partitions, there is no speed gain to having the partition, in fact the query will be slower because of having to scan all partitions.

27.8 DEFAULT Partition

What happens when you try to insert a record that can't fit in any partition?

```
INSERT INTO employees_list (birth_date, country_code) VALUES
('2001-01-01', 'JP');
ERROR: Partition key of the failing row contains (country_code) = (JP).no
partition of relation "employees_list" found for row
ERROR: no partition of relation "employees_list" found for row
SQL state: 23514
Detail: Partition key of the failing row contains (country_code) = (JP).
```

I get an error because none of my partitions are configured to accept a country_code of JP, so the insert fails because PostgreSQL does not know where to insert the row. A solution would be to add a new partition for each failure but this might not be very scalable. Better to add a default partition that accepts values not specified in the other partitions.

employees_list

> Columns

> Constraints

> Indexes

> Partitions (3)

> employees_list_default

> employees_list_eu

> employees_list_us

> RLS Policies

> Rules

> Triggers

```
CREATE TABLE employees_list_default PARTITION OF employees_list DEFAULT;
```

```
INSERT INTO employees_list (birth_date, country_code) VALUES
('2001-01-01', 'JP');
INSERT 1
Query returned successfully in 33 msec.
```

```
SELECT * FROM employees_list;
```

```
SELECT * FROM employees_list_default;
```

	id bigint	birth_date date	country_code character varying (2)
1	6	2001-01-01	JP

	id bigint	birth_date date	country_code character varying (2)
1	3	2000-12-31	UK
2	1	2000-01-01	US
3	4	2001-01-01	US
4	6	2001-01-01	JP

27.9 Multi-Level Partitioning

A single partition can also be a partition table!

Caution: When you create multi level partitions, especially on ranges, it is easy to create a large number of sub partitions, some containing little or no data.

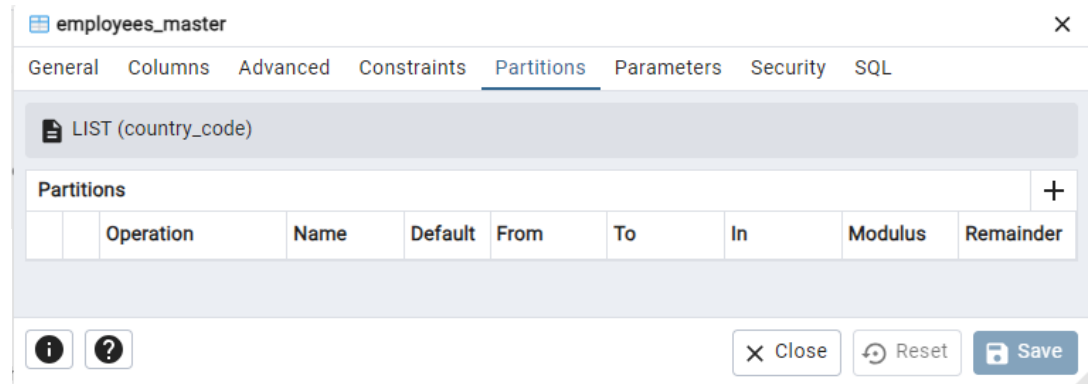
This can add many entries to the system tables, which increases the time and memory required to optimise and execute queries. In such cases, try to increase the range interval or chose a different partitioning strategy to reduce the number of sub partitions created.

Sub partition naming convention plays a vital role here so plan the naming convention.

```
Parent Table
  Partition 1
    Partition 1.1
    partition 1.2
    ...
  partition 2
    partition 2.1
    partition 2.2
    ...
```

```
CREATE TABLE employees_master (
  id BIGSERIAL,
  birth_date DATE NOT NULL,
  country_code VARCHAR(2) NOT NULL
) PARTITION BY LIST (country_code);
```

Starting with a new master table. Note how in pgAdmin4 table → properties → partitions I can see that it is partitioned by a LIST on column 'country_code'



```
CREATE TABLE employees_list_us PARTITION OF employees_list
FOR VALUES IN ('US');
```

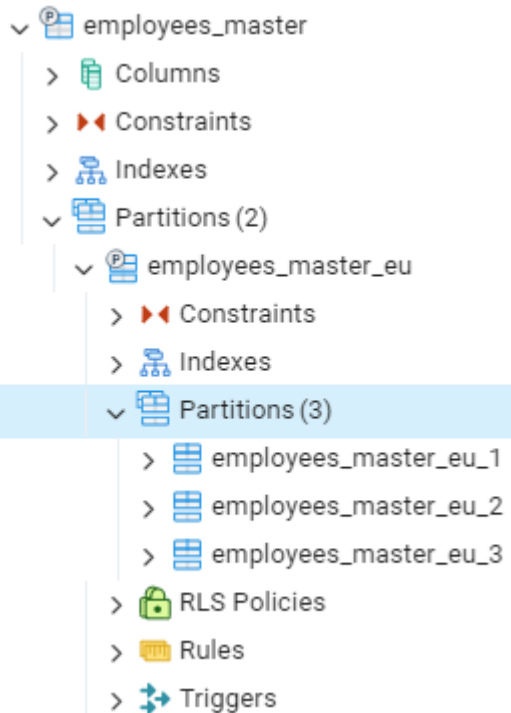
```
CREATE TABLE employees_master_eu PARTITION OF employees_list
FOR VALUES IN ('UK', 'DE', 'IT', 'FR', 'ES')
PARTITION BY HASH (id);
```

```
CREATE TABLE employees_master_eu_1 PARTITION OF employees_master_eu
FOR VALUES WITH (MODULUS 3, REMAINDER 0);
```

```
CREATE TABLE employees_master_eu_2 PARTITION OF employees_master_eu
FOR VALUES WITH (MODULUS 3, REMAINDER 1);
```

```
CREATE TABLE employees_master_eu_3 PARTITION OF employees_master_eu
FOR VALUES WITH (MODULUS 3, REMAINDER 2);
```

As before I create two partitions, one for US country_code and one for EU country_codes. BUT in the EU partition I sub-partition by HASH.



Employees_master (Parent Table)

employees_master_us (Partition 1 by LIST)

employees_master_eu (partition 2 by LIST)

employees_master_eu_1 (sub-partition 2.1 by HASH)

employees_master_eu_2 (sub-partition 2.2 by HASH)

employees_master_eu_3 (sub-partition 2.3 by HASH)

...

```
INSERT INTO employees_master (birth_date, country_code) VALUES
('2000-01-01', 'US'),
('2000-01-02', 'US'),
('2000-12-31', 'UK'),
('2001-01-01', 'DE');
```

SELECT * FROM employees_master;

	id bigint	birth_date date	country_code character varying (2)
1	4	2001-01-01	DE
2	3	2000-12-31	UK
3	1	2000-01-01	US
4	2	2000-01-02	US

SELECT * FROM ONLY employees_master;

	id bigint	birth_date date	country_code character varying (2)

SELECT * FROM ONLY employees_master_us;

	id bigint	birth_date date	country_code character varying (2)
1	1	2000-01-01	US
2	2	2000-01-02	US

SELECT * FROM ONLY employees_master_eu;

	id bigint	birth_date date	country_code character varying (2)

SELECT * FROM employees_master_eu;

	id bigint	birth_date date	country_code character varying (2)
1	4	2001-01-01	DE
2	3	2000-12-31	UK

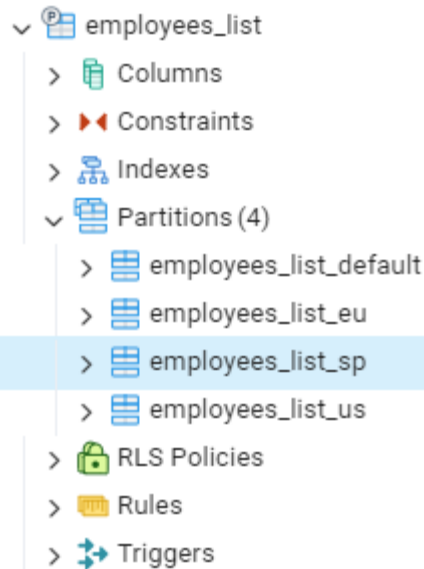
SELECT * FROM ONLY employees_master_eu_1;

	id bigint	birth_date date	country_code character varying (2)
1	4	2001-01-01	DE

SELECT * FROM ONLY employees_master_eu_2;

	id bigint	birth_date date	country_code character varying (2)
1	3	2000-12-31	UK

27.10 Partition Maintenance



Lets see we have a new country code: SP for Singapore, so we want to add a new partition to our existing master table `employees_list`

```
CREATE TABLE employees_list_sp PARTITION OF employees_list
FOR VALUES IN ('SP');
```

```
INSERT INTO employees_list (id, birth_date, country_code) VALUES
(10, '2000-01-01', 'SP');
```

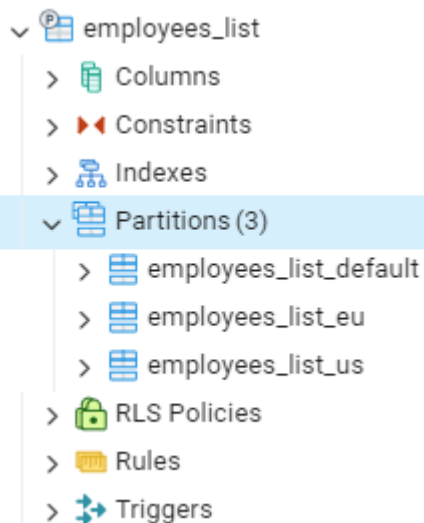
```
INSERT INTO employees_list (id, birth_date, country_code) VALUES
(10, '2000-01-01', 'SP');
```

```
SELECT * FROM employees_list;
```

	id bigint	birth_date date	country_code character varying (2)
1	3	2000-12-31	UK
2	10	2000-01-01	SP
3	1	2000-01-01	US
4	4	2001-01-01	US

```
SELECT * FROM employees_list_sp;
```

	id bigint	birth_date date	country_code character varying (2)
1	10	2000-01-01	SP



```
ALTER TABLE employees_list DETACH PARTITION employees_list_sp;
```

I can detach a partition table from a master table.

27.11 ALTERing the Partition Bounds

```
CREATE TABLE t1 (a INT, b INT) PARTITION BY RANGE (a);
CREATE TABLE t1p1 PARTITION OF t1 FOR VALUES FROM (0) TO (100);
CREATE TABLE t1p2 PARTITION OF t1 FOR VALUES FROM (200) TO (300);
INSERT INTO t1 (a,b) VALUES (1,1);
SELECT * FROM t1;
SELECT * FROM t1p1;
SELECT * FROM t1p2;
```

	a integer	b integer
1	1	1

	a integer	b integer
1	1	1

	a integer	b integer
1	1	1

```
INSERT INTO t1 (a,b) VALUES (150,150);
ERROR: Partition key of the failing row contains (a) = (150).no partition of
relation "t1" found for row
ERROR: no partition of relation "t1" found for row
SQL state: 23514
Detail: Partition key of the failing row contains (a) = (150).
```

I am going to first detach the table then alter it by modifying the range bounds then re-attach it. I will perform this within an SQL transaction to prevent anything happening while the transaction is in progress.

```
BEGIN TRANSACTION;
  ALTER TABLE t1 DETACH PARTITION t1p1;
  ALTER TABLE t1 ATTACH PARTITION t1p1 FOR VALUES FROM (0) to (200);
COMMIT TRANSACTION;
```

If something goes wrong with the transaction use

```
ROLLBACK;
```

Note that changing bounds would cause the whole master table to be rescanned and rows to be moved to the new corresponding partition table which may be noticed by users.

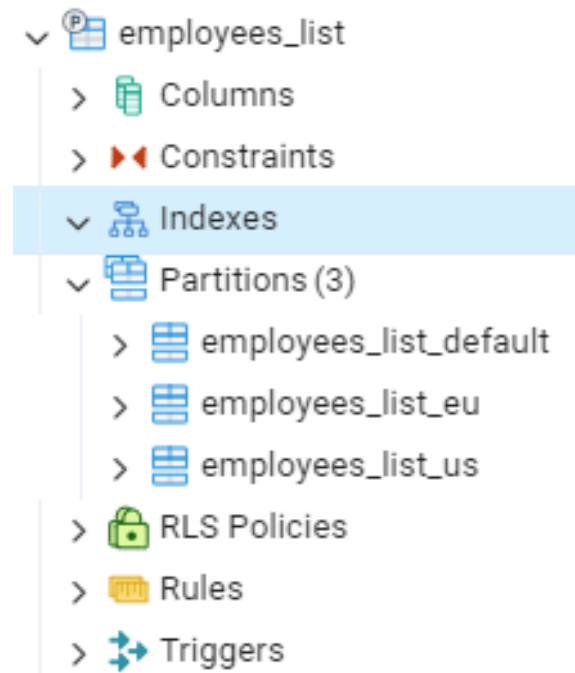
```
INSERT INTO t1 (a,b) VALUES (150,150);
INSERT 0 1
Query returned successfully in 40 msec.
```


27.12 Partition Indexes

Good: Creating an index on the master/parent table will automatically create same indexes to every partition.

Little Bad: PostgreSQL does not allow a way to create a SINGLE INDEX covering every partition of the parent table. You have to create an index for each and every partition.

The primary key, or any other unique index, **must include the columns that are used on the partition by statement.**



```
CREATE UNIQUE INDEX idx_uniq_employees_list_id ON employees_list (id);
ERROR: UNIQUE constraint on table "employees_list" lacks column "country_code" which
is part of the partition key.unique constraint on partitioned table must include all
partitioning columns
ERROR: unique constraint on partitioned table must include all partitioning columns
SQL state: 0A000
Detail: UNIQUE constraint on table "employees_list" lacks column "country_code" which
is part of the partition key.
```

employees_list

> Columns

> Constraints

> Indexes (1)

idx_uniq_employees_list_country_code

> Partitions (3)

employees_list_default

> Constraints

> Indexes (1)

employees_list_default_id_country_code_idx

> RLS Policies

> Rules

> Triggers

employees_list_eu

> Constraints

> Indexes (1)

employees_list_eu_id_country_code_idx

> RLS Policies

> Rules

> Triggers

employees_list_us

> Constraints

> Indexes (1)

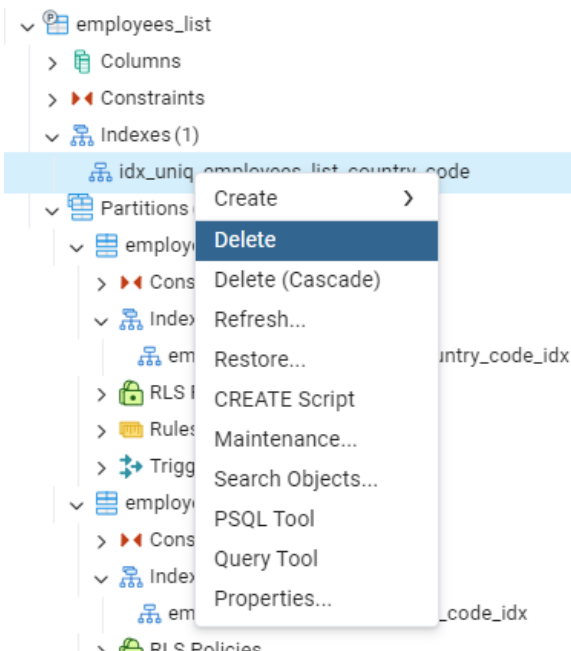
employees_list_us_id_country_code_idx

> RLS Policies

```
CREATE UNIQUE INDEX idx_uniq_employees_list_country_code
ON employees_list (id,country_code);
CREATE INDEX
Query returned successfully in 59 msec.
```

Creating an index on the master/parent table will automatically create same indexes to every partition.

PostgreSQL asks that the index be put on the partitioning column because each partition is treated as an individual table and so it adds uniqueness.



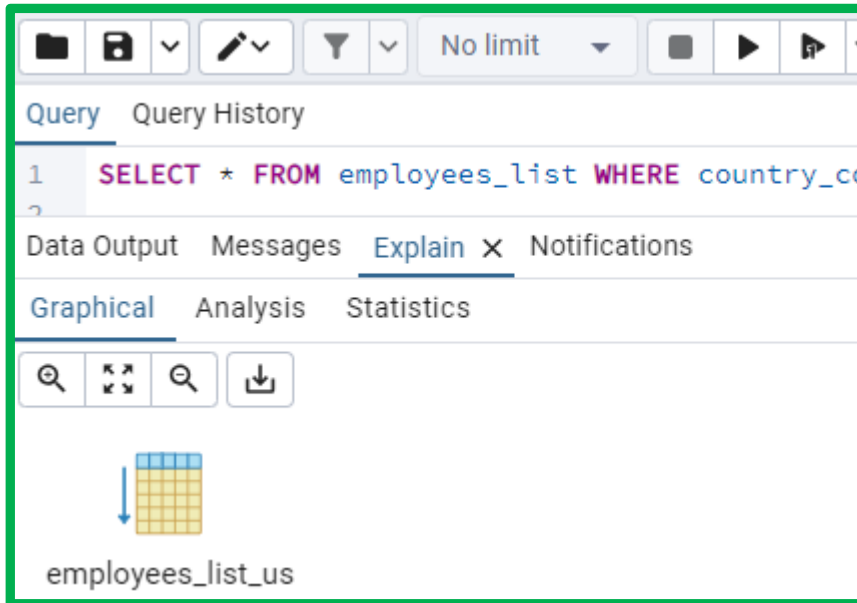
Deleting the index on the master/parent table will cause the index to be removed from all child tables.

27.13 Partition Pruning

```
SHOW enable_partition_pruning;
```

To view the current status of partition pruning

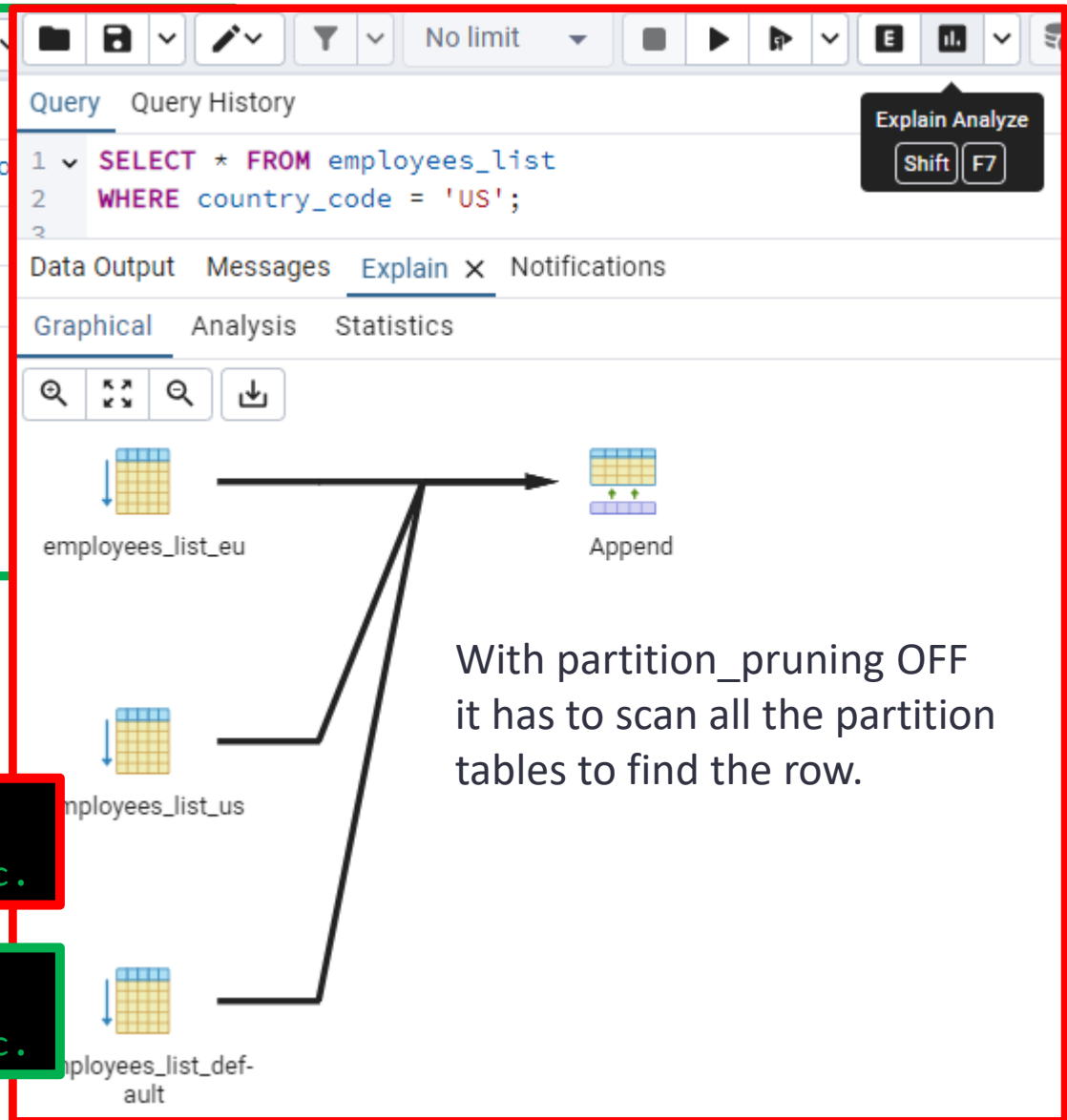
	enable_partition_pruning text
1	on



When partition pruning is on, then the partition key is used to identify which partition(s) the query should scan.

```
SET enable_partition_pruning = off;  
SET  
Query returned successfully in 71 msec.
```

```
SET enable_partition_pruning = on;  
SET  
Query returned successfully in 54 msec.
```



27.14 Determining a field for Partition Over

1. Review and understand raw data

- Get a good history, if not ALL history for ACTUAL and sample data.
- Understand the upper and lower level ranges.
- Possible queries that a user will need from this data.
- Make sure to consider this data to be added with other data sets (i.e. Joining tables).

2. Review all the possible queries

- List all queries for a table, multi-joins etc.
- Rank queries in terms of simple to advanced.
- Put advanced queries on top to consider.

3. Collect common fields in each query

- As you review the queries you will note common fields that are always used to filter rows by.

4. Rank each common field so that you can see the top 10 keys

RANK	FIELD
----	-----
1	order_date
2	country_code
3	status
...	...

5. Analyse the range of the data on each of the above keys

HIGH	LOW	RANGE
----	---	-----
2000-01-01	2020-10-10	2020-01-01 : 2020-10-10
Country_code		Global (AF US)
...		

6. Pick your best candidates for a key field to partition over from the top 5-10 fields

- Run the keys in a test environment to see which one performs best.

7. Consider what fields are being used in WHERE clauses

- For queries against the table as part of deciding which field is the right one to split it over.

27.15 Sizing the Partition

Sizing the partition is very important and knowing the minimum and maximum values is a good place to start looking for a candidate key field to partition on.

```
SELECT  
MIN(order_date),  
MAX(order_date)  
FROM orders;
```

Get unique values of key field to understand how many partitions could be needed.

```
SELECT  
DISTINCT(order_date)  
FROM orders;
```

```
SELECT  
DISTINCT(country_code)  
FROM orders;
```

Get total counts for potential key values to see how the data is distributed to determine if partitioning needs to be done by day, week, month, quarter, year, bi-annual etc...

```
SELECT  
order_date,  
Count(*)  
FROM orders  
GROUP BY order_date;
```

```
SELECT  
country_code,  
Count(*)  
FROM orders  
GROUP BY country_code;
```

Estimate the partition size based on each type of partition: Range, List & Hash.

Allow for future expansion and growth of the table. i.e. review average daily insert operations.
Monthly data = daily insertions x 28 days.

Calculate for data needs of at least 2 to 5 years into the future and create the partition accordingly.

27.16 Partitioning Advantages

The average number of index blocks you have to navigate in order to find a row goes down.

No sequential scan anymore. Having smaller blocks of your data might alter when the database can consider a sequential scan of a range a useful technique.

You can drop an individual partition, to erase all of the data from that range.

Pruning historical partition, to erase all of the data from that range.

Pruning historical data out of a partitioned table avoids VACCUUM cleanup work that DELETE leaves behind.

Another advantage is that REINDEX operations will happen in a fraction of the time vs. a single giant index to build.

Your INSERT / UPDATE / DELETE operations will be fast.

Bulk uploads of data will be quite fast in partitions.

27.17 Common Partitioning Mistakes

1. 'constraint_exclusion' is not enabled

- This cause to always include every partition for all queries. Long queries.....

2. Missing indexes on partition

- Try to replicate the same indexes that are there on the master/parent table to all partitions too.
- Make sure all the parent constraints are also added to partition tables.

3. Missing or mismatched security permissions on partition tables

- Forgetting to assign the same permissions to each child table as the parent

4. Queries not using partition field at all

- This might be the big mistake!
- Writing queries that don't filter on the partitioned key field.
- Always make sure the WHERE clause needs to filter on constraints.
- For parameterised queries, they will therefore not work right at all; And in case of functions, say those that change their value, like say CURRENT_DATE(), will also optimise poorly.

5. In General, **Keep the where clause as simple as possible**, to improve the odds that the optimiser will construct the exclusion proof you are looking for.

6. **Query overhead for partitioning is proportional to the number of partitions.** It is barely noticeable with only 10 or 20 partitions but if you have hundreds, you can expect it to be a significant impact on your database.

28. Server Programming

[28.1 PostgreSQL as a Development Platform?](#)

[28.2 Procedural Languages](#)

[28.3 Keep the Data on the Server!](#)

[28.4 Functions vs Stored Procedures](#)

[28.5 User-Defined Functions](#)

[28.6 Structure of a function](#)

28.1 PostgreSQL as a Development Platform?

Cost of acquisition of database program:

- Oracle Free edition → Enterprise sales → licensing cost
- Microsoft SQL Visible prices but with variation based on your database size/design etc.
- Open source No licensing cost, No service contract cost, **Free to use**.
Examples include **MySQL** and **PostgreSQL**.

You can try PostgreSQL for free with no financial cost for failure.

Developer Community:

- Do you have any experience with PostgreSQL?
- Good developer community from simple to advanced users.
- Regular good updates: currently on v17.

Predictability:

- A stricter adherence to SQL standards comes at the cost of not allowing ambiguous behaviour. Not allowing ambiguous behaviour makes the developer's life more difficult.
- PostgreSQL developers don't have to say "Oops, I didn't think of that," very often.

Community:

Commercial Databases (Oracle, Microsoft)	Paid Community Access
Open Source	Core developers, and vast majority of developers available at no cost.

28.2 Procedural Languages

1. Procedural languages allow us to do server programming.
2. Procedural languages see databases as 'data buckets'.
3. PostgreSQL is capable of executing server side code.
4. The user can create functions in the following built-in-programming languages:
 - SQL – (standard query language)
 - PL/pgSQL (procedural language)
 - C (for very advanced c programmers!)
5. Additional Language supports are:
 - PL/Python
 - PL/Perl
 - PL/tcl
 - PL/Java
6. The languages can be added or removed from the running version of PostgreSQL.
7. Any function defined using the above languages can also be created or dropped while PostgreSQL is running without any downtime.
8. Languages have full access to PostgreSQL internal functions, data entities → previsioning
9. Software developers are not required to learn new development languages in order to write stored procedures.

Almost anyone can code in (almost) any language they chose!

28.3 Keep the Data on the Server!

1. Server vs application layer
2. The data will be much HAPPIER if you keep it at the server end, and the performance will be much better when modifying data.
3. If you chose to do the above at the application end, then the data will need to be:
 - Fetch data from the server
 - Make modifications at application layer and
 - Then, send back modified data to the server database
4. Do not forget the 'scalability', i.e. what if you need to perform thousands or millions of customer data and process the data at the same time. Application end not sure?
5. Transactional integrity. E.g. what happens if the user account balance changes from some other transaction between the query and the update?
7.
 - Operate on columns (not on rows)
 - leave the data on the server
 - Let the database do the transactional work (that is what databases are made for!!!!)

28.4 Functions vs Stored Procedures

1. The term stored procedure has been traditionally used to talk about a function

2. A function is a:

- Normal SQL statement
- Not allowed to start or commit transactions

E.G.:

Select myfunc(id) FROM large_table;

- If we call myfunc(id) 10 million times with COMMIT?
- It is impossible to simply end the transaction in the middle of a query and launch a new one. The entire concept of transactional integrity, consistency, and so on will be violated.

3. A procedure is;

- Able to control transactions and
- Even run multiple transactions one after the other
- You cannot run it inside a SELECT statement, you have to CALL it - CALL procedure_name

4. Functions have been around for quite some time, but procedures were introduced into PostgreSQL v11.

28.5 User-Defined Functions

1. The ability to write user-defined functions is the POWERHOUSE feature of PostgreSQL!!!!

2. Functions

- Can be written in many different programming languages
- Can use any kind of control structure that the language provides
- For 'untrusted' languages, they can perform any operation that is available in PostgreSQL.

3. Functions is our way to EXTEND the functionality of PostgreSQL and beyond! A basic building block for extending PostgreSQL.

4. A function accepts input in the form of parameters, and it can create output in the form of output parameters or return values.

5. The function that we create will have the same privileges and ability that the built-in-functions possess.

6. What you do within the function is entirely up to you! TOTAL EMPOWERMENT!

28.6 Structure of a function

```
CREATE FUNCTION function_name (p1 type, p2 type, p3 type, ..... pn type)
RETURNS return_type AS
BEGIN
    -- function logic
END;
LANGUAGE language_name
```

A function is defined with the above structure.

CREATE FUNCTION	Specify the name of the function after the CREATE FUNCTION keywords
P1 type..	Make a list of parameters separated by comas.
RETURNS	Specify the return data type after the RETURNS keyword.
BEGIN / END	For the PL/pgSQL language in which the functions are written
LANGUAGE	Define the language in which the function was written (sql, plpgsql, plperl, plpython ... etc)

29. Functions with SQL Language

[29.1 Creating our First SQL Function](#)

[29.2 Introducing Dollar quoting](#)

[29.3 Function Returning no Values](#)

[29.4 Function Returning Single Value \(part 1\)](#)

[29.5 Function Returning Single Value \(part 2\)](#)

[29.6 Function Returning Single Value \(part 3\)](#)

[29.7 Function Using Parameters \(part 1\)](#)

[29.8 Function Using Parameters \(part 2\)](#)

[29.9 Function Using Parameters \(part 3\)](#)

[29.10 Function Using Parameters \(part 4\)](#)

[29.11 Function Returning a Composite](#)

[29.12 Function returning multiple rows](#)

[29.13 Function Returning a Table](#)

[29.14 Function – Order Matters!](#)

[29.15 Function as a Table Source](#)

[29.16 Function parameters with default Values](#)

[29.17 Function Based on Views](#)

[29.18 Drop a Function](#)

29.1 Creating our First SQL Function

SQL functions are the easiest to write functions in postgreSQL. We can use any SQL command inside them. The structure is simple starting with a CREATE, then RETURNS, then the SQL command inside single quotes and finally the Language.

```
CREATE OR REPLACE FUNCTION function_name() RETURNS void AS
,
-- SQL COMMAND
,
LANGUAGE SQL
```

```
CREATE OR REPLACE FUNCTION fn_mysum(INT, INT) RETURNS INT AS
,
SELECT $1 + $2;
,
LANGUAGE SQL

CREATE FUNCTION
Query returned successfully in 45 msec.
```

The dollar refers to the variables that are input to the function. In this case \$1 refers to the first INT and \$2 refers to the second INT.

- ▼ Databases (9)
 - > barcjkl_barcelonatravelhacks10
 - > basic_postgresql_course
 - > hr
 - ▼ learning
 - > Casts
 - > Catalogs
 - > Event Triggers
 - > Extensions
 - > Foreign Data Wrappers
 - > Languages
 - > Publications
 - ▼ Schemas (1)
 - ▼ public
 - > Aggregates
 - > Collations
 - > Domains
 - > FTS Configurations
 - > FTS Dictionaries
 - > Aa FTS Parsers
 - > FTS Templates
 - > Foreign Tables
 - > Functions (71)
 - > Materialized Views
 - > Operators
 - ▼ Procedures

In PgAdmin4 → Database → functions → fn_mysum() I can find the function. Note that the other functions are PostgreSQL built-in functions.

- > Domains
- > FTS Configurations
- > FTS Dictionaries
- > Aa FTS Parsers
- > FTS Templates
- > Foreign Tables
- ▼ Functions (71)
 - {=} akeys(hstore)
 - {=} avals(hstore)
 - {=} defined(hstore, text)
 - {=} delete(hstore, hstore)
 - {=} delete(hstore, text)
 - {=} delete(hstore, text[])
 - {=} each(hs hstore, OUT key text, OU
 - {=} exist(hstore, text)
 - {=} exists_all(hstore, text[])
 - {=} exists_any(hstore, text[])
 - {=} fetchval(hstore, text)
 - {=} fn_mysum(integer, integer)
 - {=} ghstore_compress(internal)
 - {=} ghstore_consistent(internal, hsto
 - {=} ghstore_decompress(internal)
 - {=} ghstore_in(cstring)
 - {=} ghstore_options(internal)
 - {=} ghstore_out(ghstore)
 - {=} ghstore_penalty(internal, internal

fn_mysum(integer, integer) X

General Definition Code Options Parameters Security SQL

Namefn_mysum

Ownerpostgres X | v

Schemapublic | v

Comment

i ?

X Close

↺ Reset

Save

fn_mysum(integer, integer) X

General Definition Code Options Parameters Security SQL

1

2SELECT \$1 + \$2;

3

i ?

X Close

↺ Reset

Save

fn_mysum(integer, integer) X

General Definition Code Options Parameters Security SQL

Return typeinteger

Languagesql X | v

Arguments

	Data type	Mode	Argument name	Default
🗑	integer v	IN v		
🗑	integer v	IN v		

i ?

X Close

↺ Reset

Save

If I right click on the function and select properties I get a number of tabs

General: function name, function owner, schema

Definition: language i.e. sql, return type, and arguments i.e. the variables for the function inputs.

```
SELECT fn_mysum();  
ERROR: function fn_mysum() does not exist  
LINE 1: SELECT fn_mysum();  
                ^  
HINT: No function matches the given name and argument types. You might need to add  
explicit type casts.  
SQL state:  
42883 Character: 8
```

TO use a function I have to call it. Not the above error message is a little useless because the error description provided irrelevant information. The function exists and the reason for the error is that I have not included the arguments/variables.

```
SELECT fn_mysum(10,20);
```

	fn_mysum integer
1	30

```
SELECT fn_mysum(1,2);
```

	fn_mysum integer
1	3

Note how it is returning an integer as we specified when creating the function.

Noteworthy is that when we created the function, the execution code i.e. `'SELECT $1 + $2;'` Is within single quotes meaning that it is interpreted as a string. This is the core of PostgreSQL's power in that we can put whatever we want inside the quotes and it will always be parsed as a string.

29.2 Introducing Dollar quoting

Text can sometimes contain quite characters for example, `SELECT 'I'm a string constant';`
Note how the word I'm contains a single quote so to use it I would have to escape the single quote `'I''m a string constant';` by using another quote.

```
CREATE OR REPLACE FUNCTION fn_mysum(INT, INT) RETURNS INT AS
$$
SELECT $1 + $2;
$$
LANGUAGE SQL
```

To solve this I can change the function to automatically escape everything that goes to our function as a string by using the double dollar sign.

In some other languages like Bash or perl, the \$ is used to identify something else like a process id so the double dollar can be changed to \$body\$

```
CREATE OR REPLACE FUNCTION fn_mysum(INT, INT) RETURNS INT AS
$body$
SELECT $1 + $2;
$body$
LANGUAGE SQL
```

29.3 Function Returning no Values

To create a function that executes but does not return values use RETURNS void

For example in the Northwinds database I have a table called employees which has a column called country and I want a function that will replace country values with 'n/a'

```
SELECT * from employees;
```

	code character varying (10)	country character varying (15)	home_phone character varying (24)	extension character varying (4)	photo bytea	notes text
1		USA	(206) 555-9857	5467	[binary d...]	Education includes a BA in psychology from Colorado State
2		USA	(206) 555-9482	3457	[binary d...]	Andrew received his BTS commercial in 1974 and a Ph.D. in
3		USA	(206) 555-3412	3355	[binary d...]	Janet has a BS degree in chemistry from Boston College (1

```
CREATE OR REPLACE FUNCTION fn_employees_update_countries() RETURNS VOID AS
$$
UPDATE employees SET country = 'N/A' WHERE country IS NULL;
$$
LANGUAGE SQL
```

The above function does not take any parameters and returns void. The function will set the country to 'N/A' for all rows where country is null.

```
SELECT fn_employees_update_countries();
```

When I call the function, the return is void.

	fn_employees_update_countries void
1	[null]

```
UPDATE employees SET country = NULL WHERE employee_id = 1;
```

```
SELECT * from employees;
```

	id (15)	region character varying (15)	postal_code character varying (10)	country character varying (15)	home_phone character varying (24)	extension character varying (4)	photo bytea
1		WA	98401	USA	(206) 555-9482	3457	[binary]
2		WA	98033	USA	(206) 555-3412	3355	[binary]
3		WA	98052	USA	(206) 555-8122	5176	[binary]
4		[null]	SW1 8JR	UK	(71) 555-4848	3453	[binary]
5		[null]	EC2 7JR	UK	(71) 555-7773	428	[binary]
6		[null]	RG1 9SP	UK	(71) 555-5598	465	[binary]
7		WA	98105	USA	(206) 555-1189	2344	[binary]
8		[null]	WG2 7LT	UK	(71) 555-4444	452	[binary]
9		WA	98122	[null]	(206) 555-9857	5467	[binary]

```
SELECT fn_employees_update_countries();
```

```
SELECT * from employees;
```

If I update one of the rows to have a null country then call the function I can see the result is that the function has changed the null value to 'N/A'

(Housekeeping)

```
UPDATE employees
SET country = 'USA'
WHERE employee_id = 1;
```

postal_code character varying (10)	country character varying (15)	home_phone character varying (24)	extension character varying (4)	photo bytea
98401	USA	(206) 555-9482	3457	[binary]
98033	USA	(206) 555-3412	3355	[binary]
98052	USA	(206) 555-8122	5176	[binary]
SW1 8JR	UK	(71) 555-4848	3453	[binary]
EC2 7JR	UK	(71) 555-7773	428	[binary]
RG1 9SP	UK	(71) 555-5598	465	[binary]
98105	USA	(206) 555-1189	2344	[binary]
WG2 7LT	UK	(71) 555-4444	452	[binary]
98122	N/A	(206) 555-9857	5467	[binary]

29.4 Function Returning Single Value (part 1)

```
SELECT * FROM products ORDER BY unit_price;
```

	product_id [PK] smallint	product_name character varying (40)	supplier_id smallint	category_id smallint	quantity_per_unit character varying (20)	unit_price real	units_in_stock smallint
1	33	Geitost	15	4	500 g	2.5	112
2	24	Guaraná Fantástica	10	1	12 - 355 ml cans	4.5	20
3	13	Konbu	6	8	2 kg box	6	24

The syntax is the same as before but I must specify a return type. Note how the unit price is 'real'

```
CREATE OR REPLACE FUNCTION fn_min_unit_price() RETURNS INTEGER AS  
$$  
SELECT MIN(unit_price) from products;  
$$  
LANGUAGE SQL  
CREATE FUNCTION  
Query returned successfully in 73 msec.
```

I create the function that is returning an 'integer'

```
SELECT fn_min_unit_price();
```

Note how the data in the table is a 'real' type with a value of 2.5 but because I have set it to integer, it is returning a whole number, i.e. 2

	fn_min_unit_price integer
1	2


```
CREATE OR REPLACE FUNCTION fn_min_unit_price() RETURNS REAL AS
$$
SELECT MIN(unit_price) from products;
$$
LANGUAGE SQL
ERROR: cannot change return type of existing function
HINT: Use DROP FUNCTION fn_min_unit_price() first.
SQL state: 42P13
```

```
DROP FUNCTION fn_min_unit_price();
DROP FUNCTION
Query returned successfully in 49 msec.
```

```
CREATE OR REPLACE FUNCTION fn_products_min_price() RETURNS REAL AS
$$
SELECT MIN(unit_price) from products;
$$
LANGUAGE SQL
```

```
SELECT fn_products_min_price();
```

	fn_products_min_price
1	2.5

In v17 of PostgreSQL it will not let me change the return type but instead suggests that I drop the function first.

When I create the function to return the correct type then call the function I now get the actual value of 2.5.

```
CREATE OR REPLACE FUNCTION fn_products_max_price() RETURNS REAL AS
$$
SELECT MAX(unit_price) FROM products;
$$
LANGUAGE SQL
```

```
SELECT fn_products_min_price();
```

	fn_products_max_price
1	263.5

A function to get the max price.

Both the above functions return a **SINGLE VALUE**

```
SELECT * FROM order_details;
```

	order_id [PK] smallint	product_id [PK] smallint	unit_price real	quantity smallint	discount real
1	10248	11	14	12	0
2	10248	42	9.8	10	0
3	10248	72	34.8	5	0
4	10249	14	18.6	9	0
5	10249	51	42.4	40	0
6	10250	41	7.7	10	0

```
SELECT
order_id,
SUM(unit_price * quantity) AS amount
FROM order_details;
GROUP BY order_id
ORDER BY 2 DESC
LIMIT 1;
```

	order_id smallint	amount double precision
1	10865	17250

The aim is to get the biggest order from the order_details table where an order is made up of one or more rows (note the same order_id's) containing a unit price and quantity. Note that when testing the query, it returns 'double precision' type.

```
CREATE OR REPLACE FUNCTION fn_biggest_order() RETURNS DOUBLE PRECISION AS
$$
SELECT
MAX(amount) FROM
(
SELECT
order_id,
SUM(unit_price * quantity) AS amount
FROM order_details
GROUP BY order_id
) as total_amount
$$
LANGUAGE SQL
```

```
SELECT fn_biggest_order();
```

	fn_biggest_order double precision
1	17250

Now when I create the function I ensure to specify the correct return type and also tidy up the query to use a sub query instead of limit clause. Calling the function returns a single value of the largest order.

```
CREATE OR REPLACE FUNCTION fn_smallest_order() RETURNS DOUBLE PRECISION AS
$$
SELECT
MIN(amount) FROM
(
SELECT
order_id,
SUM(unit_price * quantity) AS amount
FROM order_details
GROUP BY order_id
) AS total_amount
$$
LANGUAGE SQL
```

```
SELECT fn_smallest_order();
```

	fn_smallest_order	
	double precision	
1	12.5	

29.5 Function Returning Single Value (part 2)

I want 3 functions that do the following:

Get total count of customers, Get total count of products, Get total count of orders.

```
CREATE OR REPLACE FUNCTION fn_api_get_total_customers() RETURNS BIGINT AS
$$
SELECT COUNT(*) FROM customers;
$$
LANGUAGE SQL
```

```
SELECT fn_api_get_total_customers();
```

	fn_api_get_total_customers	
	bigint	
1	91	

```
CREATE OR REPLACE FUNCTION fn_api_get_total_products() RETURNS BIGINT AS
$$
SELECT COUNT(*) FROM products;
$$
LANGUAGE SQL
```

```
SELECT fn_api_get_total_products();
```

	fn_api_get_total_products	
	bigint	
1	77	

```
CREATE OR REPLACE FUNCTION fn_api_get_total_orders() RETURNS BIGINT AS
$$
SELECT COUNT(*) FROM orders;
$$
LANGUAGE SQL
```

```
SELECT fn_api_get_total_orders();
```

	fn_api_get_total_orders	
	bigint	
1		830

29.6 Function Returning Single Value (part 3)

```
CREATE OR REPLACE FUNCTION fn_api_total_customers_empty_fax() RETURNS BIGINT AS
$$
SELECT COUNT(*) FROM customers WHERE fax IS NULL;
$$
LANGUAGE SQL
```

```
SELECT fn_api_total_customers_empty_fax();
```

	fn_api_total_customers_empty_fax	
	bigint	
1		22

```
CREATE OR REPLACE FUNCTION fn_api_total_customers_empty_region() RETURNS BIGINT AS
$$
SELECT COUNT(*) FROM customers WHERE region IS NULL;
$$
LANGUAGE SQL
```

```
SELECT fn_api_total_customers_empty_region();
```

	fn_api_total_customers_empty_region	
	bigint	
1		60

29.7 Function Using Parameters (part 1)

P1 Type, P2 Type etc.. Function parameters/arguments can be passed and accessed by name, instead of just by ordinal order (\$1, \$2...) which makes the code much better for readability.

```
CREATE OR REPLACE FUNCTION function_name(p1 type, p2 type) RETURNS return_type AS
$$
-- $1, $2 (use the ordinal position)
-- p1, p2 (use the parameter name)
-- customer_id, company_name (Bad practice naming parameters after column_names)
-- p_param_name, p_param2_name
$$
LANGUAGE SQL
```

When creating a function that takes parameters, **I name them and specify the type in the brackets**. Then in the body of the function I can reference those parameters by their **ordinal position** or better **by their name**.

Note that it is good practice to **not name the parameters the same as column_names** in the table as this may create some errors and makes readability really poor. For example a query that wants to find customer's by id parameter would be: WHERE customer_id = customer_id;

When naming parameters it is good practice to use a standard nomenclature. For example **p_parameter_name** where the p identifies it as a parameter.

```
CREATE OR REPLACE FUNCTION fn_mid(p_string VARCHAR, p_starting_point INTEGER) RETURNS VARCHAR AS
$$
SELECT SUBSTRING(p_string, p_starting_point);
$$
LANGUAGE SQL
```

```
SELECT fn_mid('Amazing PostGresSQL', 2);
```

	fn_mid	character varying	🔒
1	mazing PostGresSQL		

```
CREATE OR REPLACE FUNCTION fn_api_get_total_customers_by_city(p_city VARCHAR) RETURNS BIGINT AS
$$
SELECT count(*) FROM customers WHERE city = p_city;
$$
LANGUAGE SQL
```

```
SELECT fn_api_get_total_customers_by_city('London');
```

	fn_api_get_total_customers_by_city	bigint	🔒
1	6		

```
CREATE OR REPLACE FUNCTION fn_api_get_total_customers_by_country(p_country VARCHAR) RETURNS
BIGINT AS
$$
SELECT count(*) FROM customers WHERE country = p_country;
$$
LANGUAGE SQL
```

```
SELECT fn_api_get_total_customers_by_country('UK');
```

	fn_api_get_total_customers_by_country	bigint	🔒
1	7		

These functions can be exposed to a client side application to return data to the application.

29.8 Function Using Parameters (part 2)

```
CREATE OR REPLACE FUNCTION fn_api_customers_total_orders(p_customer_id bpchar) RETURNS  
BIGINT AS  
$$  
SELECT count(*)  
FROM orders  
NATURAL JOIN customers  
WHERE customer_id = p_customer_id;  
$$  
LANGUAGE SQL
```

```
SELECT fn_api_customers_total_orders('VINET');
```

	fn_api_customers_total_orders	
	bigint	
1		5

29.9 Function Using Parameters (part 3)

```
SELECT  
orders.order_id,  
orders.customer_id,  
products.product_name,  
order_details.unit_price,  
order_details.quantity,  
((order_details.unit_price * order_details.quantity) - order_details.discount) as  
total_amount  
FROM orders  
NATURAL JOIN order_details  
NATURAL JOIN products  
WHERE orders.customer_id = 'ALFKI';
```

Get the biggest order for a specific customer. First I write a query to review the data.

```
CREATE OR REPLACE FUNCTION fn_api_customers_largest_order(p_customer_id bpchar) RETURNS
double precision AS
$$
SELECT
MAX(order_amount)
FROM
(
SELECT
orders.order_id,
SUM((unit_price * quantity) - discount) as order_amount
FROM order_details
NATURAL JOIN orders
NATURAL JOIN products
WHERE orders.customer_id = p_customer_id
GROUP BY orders.order_id
) total_amount
$$
LANGUAGE SQL
```

```
SELECT fn_api_customers_largest_order('ALFKI');
```

	fn_api_customers_largest_order 
1	1085.2499771118164

29.7 Function Using Parameters (part 4)

Get the most ordered product by customer – start to work out the query over several steps

```
SELECT * FROM customers
NATURAL JOIN order_details
NATURAL JOIN products
WHERE customer_id = 'CACTU';
```

```
SELECT
orders.order_id,
orders.customer_id,
products.product_name,
order_details.unit_price,
order_details.quantity
FROM orders
NATURAL JOIN order_details
NATURAL JOIN products
WHERE customer_id = 'CACTU'
ORDER BY 5 DESC;
```

```
SELECT
product_id,
SUM(quantity) AS total_quantity
FROM order_details
NATURAL JOIN orders
WHERE orders.customer_id = 'CACTU'
GROUP BY product_id
ORDER BY 2 DESC
LIMIT 1;
```



```

CREATE OR REPLACE FUNCTION fn_api_customers_most_ordered_product(p_customer_id bpchar)
RETURNS VARCHAR AS
$$
SELECT
product_name
FROM products
WHERE product_id IN
(
SELECT
product_id
FROM
(
SELECT
product_id,
SUM(quantity) AS total_quantity
FROM order_details
NATURAL JOIN orders
WHERE orders.customer_id = p_customer_id
GROUP BY product_id
ORDER BY 2 DESC
LIMIT 1
) AS product_order
)
$$
LANGUAGE SQL

```

```
SELECT fn_api_customers_most_ordered_product('CACTU');
```

	fn_api_customers_most_ordered_product 
1	Sasquatch Ale

29.11 Function Returning a Composite

RETURNS table_name.

```
CREATE OR REPLACE FUNCTION fn_api_order_latest() RETURNS orders AS
$$
SELECT * FROM orders ORDER BY order_date DESC, order_id DESC LIMIT 1;
$$
LANGUAGE SQL
```

Returns a single row, in the form of an array style where each item in the array is separated by a coma.

```
SELECT fn_api_order_latest();
```

	fn_api_order_latest orders
1	(11077,RATTC,1,1998-05-06,1998-06-03,,2,8.53,"Rattlesnake Canyon Grocery","2817 Milton Dr.",Albuquerque,NM,87110,USA)

```
SELECT (fn_api_order_latest()).*;
```

 If I append with dot star, I can get the output as a table.

	order_id smallint	customer_id character	employee_id smallint	order_date date	required_date date	shipped_date date	ship_via smallint	freight real	ship_name character varying (40)	ship_address character varying (60)
1	11077	RATTC	1	1998-05-06	1998-06-03	[null]	2	8.53	Rattlesnake Canyon Grocery	2817 Milton Dr.

```
SELECT (fn_api_order_latest()).customer_id;
```

If I append with dot column_name I can get a specific column

	customer_id character
1	RATTC

```
SELECT customer_id(fn_api_order_latest());
```

I can also get a specific column by naming it before calling the function

	customer_id character
1	RATTC

```
CREATE OR REPLACE FUNCTION fn_api_order_latest_by_date_range(p_from DATE, p_to DATE)
RETURNS orders AS
$$
SELECT
*
FROM
orders
WHERE order_date BETWEEN p_from AND p_to
ORDER BY order_date DESC, order_id DESC
LIMIT 1;
$$
LANGUAGE SQL
```

```
SELECT (fn_api_order_latest_by_date_range('1997-01-01', '2020-10-10')).*;
```

	order_id smallint	customer_id character	employee_id smallint	order_date date	required_date date	shipped_date date	ship_via smallint	freight real	ship_name character varying (40)	s c
1	11077	RATTC	1	1998-05-06	1998-06-03	[null]	2	8.53	Rattlesnake Canyon Grocery	2

```
CREATE OR REPLACE FUNCTION fn_api_employee_latest_hirep_from() RETURNS employees AS
$$
SELECT * FROM employees ORDER BY hire_date DESC LIMIT 1;
$$
LANGUAGE SQL
```

```
SELECT (fn_api_employee_latest_hirep_from()).*;
```

	employee_id smallint	last_name character varying (20)	first_name character varying (10)	title character varying (30)	title_of_courtesy character varying (25)	birth_date date	hire_date date
1	9	Dodsworth	Anne	Sales Representative	Ms.	1966-01-27	1994-11-15

29.12 Function Returning Multiple Rows

```
CREATE OR REPLACE FUNCTION fn_api_employees_hire_by_year(p_year INTEGER) RETURNS SETOF
employees AS
$$
SELECT * FROM employees WHERE EXTRACT('year' FROM hire_date) = p_year;
$$
LANGUAGE SQL
```

```
SELECT (fn_api_employees_hire_by_year(1992)).*;
```

	employee_id smallint	last_name character varying (20)	first_name character varying (10)	title character varying (30)	title_of_courtesy character varying (25)	birth_date date	hire_date date	address character varying (255)
1	2	Fuller	Andrew	Vice President, Sales	Dr.	1952-02-19	1992-08-14	908 W. Capital Way, Tacoma, WA, 98401, USA
2	3	Leverling	Janet	Sales Representative	Ms.	1963-08-30	1992-04-01	722 Moss Bay Blvd., Kirkland, WA, 98033, USA
3	1	Davolio	Nancy	Sales Representative	Ms.	1948-12-08	1992-05-01	507 - 20th Ave. E., Seattle, WA, 98122, USA

To get a multiple row return from a function I use RETURNS SETOF to specify that I want a set of results.

```
SELECT fn_api_employees_hire_by_year(1992);
```

	fn_api_employees_hire_by_year employees
1	(2,Fuller,Andrew,"Vice President, Sales",Dr.,1952-02-19,1992-08-14,"908 W. Capital Way",Tacoma,WA,98401,USA,"(206) 555-9482",3457,"\\x","Andrew received his BTS com
2	(3,Leverling,Janet,"Sales Representative",Ms.,1963-08-30,1992-04-01,"722 Moss Bay Blvd.",Kirkland,WA,98033,USA,"(206) 555-3412",3355,"\\x","Janet has a BS degree in cl
3	(1,Davolio,Nancy,"Sales Representative",Ms.,1948-12-08,1992-05-01,"507 - 20th Ave. E.\\nApt. 2A",Seattle,WA,98122,USA,"(206) 555-9857",5467,"\\x","Education includes a

```

CREATE OR REPLACE FUNCTION fn_api_products_total_amount_by(p_amount INTEGER) RETURNS SETOF
products AS
$$
SELECT * FROM products WHERE product_id IN
(
SELECT product_id FROM
(
SELECT
product_id,
SUM((unit_price * quantity) - discount) AS product_total_sale
FROM order_details
GROUP BY product_id
HAVING SUM((unit_price * quantity) - discount) > p_amount
) AS t1
)
$$
LANGUAGE SQL

```

```

SELECT (fn_api_products_total_amount_by(100000)).*;

```

	product_id smallint 🔒	product_name character varying (40) 🔒	supplier_id smallint 🔒	category_id smallint 🔒	quantity_per_unit character varying (20) 🔒	unit_price real 🔒	units_in_stock smallint 🔒	units_on_o smallint
1	38	Côte de Blaye	18	1	12 - 75 cl bottles	263.5	17	

29.13 Function Returning a Table

We can use functions as a table source i.e. `SELECT Column_list FROM function_name`

In lesson [29.12](#), I created a function to select all employees stored in a given year where the function takes year as an input. I can use the output of this function like an ordinary table and select rows from it.

```
SELECT
first_name,
last_name,
hire_date
FROM fn_api_employees_hire_by_year(1992);
```

	first_name character varying (10) 🔒	last_name character varying (20) 🔒	hire_date date 🔒
1	Andrew	Fuller	1992-08-14
2	Janet	Leverling	1992-04-01
3	Nancy	Davolio	1992-05-01

```
SELECT
*
FROM
fn_api_products_total_amount_by(100000);
```

	product_id smallint 🔒	product_name character varying (40) 🔒	supplier_id smallint 🔒	category_id smallint 🔒	quantity_per_unit character varying (20) 🔒	unit_price real 🔒	units_in_stock smallint 🔒	units_on_order smallint 🔒	rec sm
1	38	Côte de Blaye	18	1	12 - 75 cl bottles	263.5	17	0	

29.14 Functions - Order Matters!

```
CREATE OR REPLACE FUNCTION fn_api_customer_top_orders(p_customer_id bpchar, p_limit
INTEGER) RETURNS TABLE
```

```
(
    order_id SMALLINT,
    customer_id BPCHAR,
    product_name VARCHAR,
    unit_price REAL,
    total_order DOUBLE PRECISION,
    quantity SMALLINT
```

```
)
AS
$$
SELECT
orders.order_id,
orders.customer_id,
products.product_name,
order_details.unit_price,
order_details.quantity
FROM orders
NATURAL JOIN order_details
NATURAL JOIN products
WHERE orders.customer_id = p_customer_id
ORDER BY ((unit_price * quantity) - discount) DESC
LIMIT p_limit
$$
```

```
LANGUAGE SQL
```

```
ERROR: Final statement returns too few columns.return type mismatch in function
declared to return record
```

```
ERROR: return type mismatch in function declared to return record
```

```
SQL state: 42P13
```

```
Detail: Final statement returns too few columns.
```

```
Context: SQL function "fn_api_customer_top_orders"
```

In this function I am specifying the rows to be output in the table but note the errors.

1- final statement returns too few columns. This is because I am specifying 6 columns but selecting 7 columns. I also have a type mismatch because the total_order column is the fifth column and in the select query the fifth column is order details quantity which is an integer.

The order of the columns in the table definition should match the order of the columns in the select query.

```
CREATE OR REPLACE FUNCTION fn_api_customer_top_orders(p_customer_id bpchar, p_limit
INTEGER) RETURNS TABLE
```

```
(
    order_id SMALLINT,
    customer_id BPCHAR,
    product_name VARCHAR,
    unit_price REAL,
    quantity SMALLINT,
    total_order DOUBLE PRECISION
```

```
)
AS
$$
SELECT
orders.order_id,
orders.customer_id,
products.product_name,
order_details.unit_price,
order_details.quantity,
((order_details.unit_price*order_details.quantity)-discount) AS total_order
FROM orders
NATURAL JOIN order_details
NATURAL JOIN products
WHERE orders.customer_id = p_customer_id
ORDER BY ((unit_price * quantity) - discount) DESC
LIMIT p_limit
$$
LANGUAGE SQL
```

```
CREATE FUNCTION
```

```
Query returned successfully in 34 msec.
```

I have corrected the function so that the first column defined in the table matched the first column in the select query in terms of data type and naming. Etc for all six columns.

29.15 Functions as a Table Source

I can return a table from a function where I specify RETURNS TABLE. See lesson [29.14](#) for the syntax. Note that by default the return is a list of arrays.

```
SELECT (fn_api_customer_top_orders('VINET', 5)).*;
```

	order_id smallint	customer_id character	product_name character varying	unit_price real	quantity smallint	total_order double precision
1	10248	VINET	Mozzarella di Giovanni	34.8	5	173.99999618530273
2	10739	VINET	Filo Mix	7	18	126
3	10737	VINET	Jack's New England Clam Chowder	9.65	12	115.79999542236328
4	10739	VINET	Inlagd Sill	19	6	114
5	10737	VINET	Konbu	6	4	24

```
SELECT fn_api_customer_top_orders('VINET', 5);
```

	fn_api_customer_top_orders record
1	10248,VINET,Mozzarella di Giovanni,34.8,5,173.99999618530273
2	10739,VINET,Filo Mix,7,18,126
3	10737,VINET,Jack's New England Clam Chowder,9.65,12,115.79999542236328
4	10739,VINET,Inlagd Sill,19,6,114
5	10737,VINET,Konbu,6,4,24

29.16 Function Parameter with Default Values

```
CREATE OR REPLACE FUNCTION function_name(p1 type DEFAULT v1, p2 type DEFAULT v2)
RETURNS return_type AS
$$
-- SQL logic
$$
LANGUAGE SQL
```

I can define default values in the input parameters of a function using the DEFAULT keyword.

```
CREATE OR REPLACE FUNCTION fn_sum_3 (x INT, y INT DEFAULT 10, z INT DEFAULT 10) RETURNS
INTEGER AS
$$
SELECT x + y + Z;
$$
LANGUAGE SQL
```

If I call the function without specifying all the values then it will use default values.

```
SELECT fn_sum_3(1,2,3);
```

	fn_sum_3 integer
1	6

```
SELECT fn_sum_3(1);
```

	fn_sum_3 integer
1	21

```
DROP FUNCTION fn_sum_3;
```

```
CREATE OR REPLACE FUNCTION fn_sum_3 (x INT, y INT DEFAULT 10, z INT) RETURNS INTEGER AS
$$
SELECT x + y + z;
$$
LANGUAGE SQL
```

If I use a default value, all other parameters after must also have a default value

ERROR: input parameters after one with a default value must also have defaults
SQL state: 42P13

So if I am defining a function that takes several values, and one of them is a default, then I will push the default one to the last position of input parameters.

```
CREATE OR REPLACE FUNCTION fn_api_new_price(products, percentage_increase NUMERIC
DEFAULT 107) RETURNS double precision AS
$$
SELECT $1.unit_price * percentage_increase / 100;
$$
LANGUAGE SQL
```

```
SELECT
product_id,
product_name,
unit_price,
fn_api_new_price(products.*) AS new_price
FROM
products;
```

I can use a function as a column in a query.

	product_id [PK] smallint	product_name character varying (40)	unit_price real	new_price double precision
1	1	Chai	18	19.26
2	2	Chang	19	20.33
3	3	Aniseed Syrup	10	10.7
4	4	Chef Anton's Cajun Seasoning	22	23.54

29.17 Function Based on Views

```
SELECT * FROM pg_stat_activity WHERE state = 'active';
```

	datid oid	datname name	pid integer	leader_pid integer	usesysid oid	username name	application_name text	client_addr inet	client_hostname text	client_port integer
1	16665	northwind	1000	[null]	10	postgres	pgAdmin 4 - CONN:9051286	::1	[null]	52629

```
CREATE OR REPLACE VIEW v_active_queries AS
SELECT
pid,
username,
query_start,
(CURRENT_TIMESTAMP - query_start) AS runtime,
query
FROM pg_stat_activity
WHERE state = 'active'
ORDER BY 4 DESC;
```

The pg_stat_activity view will have one row per server process, showing information related to the current activity of that process.

I can create a view to find long running processes. And build a function to select data from the view.

```
CREATE OR REPLACE FUNCTION fn_internal_active_queries(p_limit INT) RETURNS SETOF
v_active_queries AS
$$
SELECT * FROM v_active_queries LIMIT p_limit;
$$
LANGUAGE SQL
```

```
SELECT * FROM fn_internal_active_queries(5);
```

Calling the function based on a view

	pid integer	username name	query_start timestamp with time zone	runtime interval	query text
1	1000	postgres	2025-02-16 01:14:15.227221+01	00:00:00	SELECT * FROM fn_internal_active_queries(5);

29.18 Drop a Function

```
DROP FUNCTION [IF EXISTS] function_name(argument_list) [CASCADE|RESTRICT];
```

function_name	Try to keep the name of the function unique GLOBALLY!!
IF EXISTS	issue a notice instead of an error in case the function does not exist
Argument_list	Since functions can be overloaded, PostgreSQL needs to know what function you want to remove by checking the arguments list.
CASCADE	To drop the function and it's dependants (Be Careful!)
RESTRICT	Rejects the removal of a function when it has any dependant objects.

```
CREATE OR REPLACE FUNCTION fn_sum(x INT, y INT) RETURNS INTEGER AS  
$$  
SELECT x+y;  
$$  
LANGUAGE SQL
```

```
DROP FUNCTION fn_sum;
```

30. Functions with PL/PgSQL Language

[30.1 Introduction to PL/pgSQL Language](#)

[30.2 PL/pgSQL vs SQL](#)

[30.3 Structure of PL/pgSQL Function](#)

[30.4 PL/pgSQL Block Structure](#)

[30.5 Declaring Variables](#)

[30.6 Declaring Variables via ALIAS FOR](#)

[30.7 Declaring Variables in Function](#)

[30.8 Variable initializing Timing](#)

[30.9 Copying Data Types](#)

[30.10 Assigning Variables From Query](#)

[30.11 Using IN OUT Without Returns](#)

[30.12 Variables in Block and Sub-block](#)

[30.13 How to Return Query Results](#)

[30.14 Control Structures – IF Statement](#)

[30.15 Using IF with table data](#)

[30.16 CASE statement](#)

[30.17 Searched CASE Statement](#)

[30.18 Loop Statement](#)

[30.19 FOR Loops Iterate over Result Set](#)

[30.20 CONTINUE Statement](#)

[30.21 FOREACH Loop with Arrays](#)

[30.22 WHILE Loop](#)

[30.23 Using Return Query](#)

[30.24 Returning a Table](#)

[30.25 Using RETURN NEXT](#)

[30.26 Error and Exception Handling](#)

[30.27 Exception – Too Many Rows](#)

[30.28 Using SQL STATE codes for exception Handling](#)

[30.29 Exception with Data Exception Errors](#)

30.1 Introduction to PL/pgSQL Language

1. PL/pgSQL is a powerful SQL scripting language that is heavily influenced by Oracle PL/SQL
2. PL/pgSQL is a fully-fledged SQL development language
3. Originally designed for simple scalar functions, and now providing:
 - Full PostgreSQL internals
 - Control structure
 - Variable Declaration
 - Expression
 - Loops
 - Cursors and much more
4. Ability to Create:
 - Complex functions
 - New Data types
 - Stored procedures and more...
5. It is easy to use
6. Available by default on PostgreSQL
7. Optimised for performance of data intensive tasks

30.2 PL/pgSQL vs SQL

1. SQL is a query language but can only execute SQL statements INDIVIDUALLY
2. PL/pgSQL Will:
 - Wrap multiple statements in an OBJECT
 - Store that object on the server
 - Instead of sending multiple statements to the server one by one
 - We can send one statement to execute the 'object' stored in the server
 - Reduce round trips to server for each statement to execute
 - Provide transactional integrity and much more

30.3 Structure of a PL/pgSQL Function

```
CREATE FUNCTION function_name(p1 type, p2 type...) RETURNS return_type AS
```

```
$$
```

```
BEGIN
```

```
-- Statements
```

```
END
```

```
$$
```

```
LANGUAGE plpgsql
```

The Structure of a PL/pgSQL function has a BEGIN and END with semi-colon. In the statement, instead of SELECTING I need to use RETURN.

Note also the language is plpgsql

```
CREATE FUNCTION fn_api_products_max_price() RETURNS bigint AS
```

```
$$
```

```
BEGIN
```

```
    RETURN MAX(unit_price) FROM products;
```

```
END;
```

```
$$
```

```
LANGUAGE plpgsql
```

```
SELECT fn_api_products_max_price();
```

	fn_api_products_max_price	
	bigint	
1		264

30.4 PL/pgSQL Block Structure

```
[ <<label>> ]
```

```
[DECLARE
```

```
    Declarations ];
```

```
BEGIN
```

```
    statements;
```

```
END [ label ];
```

A PL/pgSQL function or stored procedure is organised into blocks

The block structure syntax has an optional label at the beginning and end.

The DECLARE and declarations are mandatory.

The body section must go between the BEGIN and END.

Each statement in the body ends with a semi colon.

```
DECLARE
    variable_name data_type [:= expression];
BEGIN
    statements;
END;
```

```
DECLARE
    mynum          integer :=1;
    first_name     varchar(100) :='Elliott';
    hire_Date      date :='2020-01-01';
    start_time     timestamp := NOW();
    emptyvar       integer;
```

```
BEGIN
...
END;
```

```
DO
$$
DECLARE
    mynum          integer :=1;
    first_name     varchar(100) :='Elliott';
    hire_Date      date :='2020-01-01';
    start_time     timestamp := NOW();
    emptyvar       integer;
BEGIN
    RAISE NOTICE 'My Variables % % % % %',
        mynum,
        first_name,
        hire_date,
        start_time,
        emptyvar;
END
$$
```

30.5 Declaring Variables

A variable holds a value that can be changed through the block

Before using a variable, you must declare it in the declaration section.

:= can be used to define a default value.

I can also use SQL functions as default values such as NOW().

Emptyvar does not have a default value.

```
NOTICE: My Variables 1 Elliott 2020-01-01
2025-02-17 20:04:55.161484 <NULL>
DO
Query returned successfully in 44 msec.
```

After executing this block the return is the variables declared.

Raise notice means give an SQL NOTICE output.

The block is initialising the variable then using it.

30.6 Declaring Variables via ALIAS FOR

```
newname ALIAS FOR oldname;
```

```
CREATE OR REPLACE FUNCTION function_name(int, int) RETURNS int AS  
$$
```

```
DECLARE
```

```
    x ALIAS FOR $1;
```

```
    y ALIAS FOR $2;
```

```
BEGIN
```

```
    -- Statements
```

```
END;
```

```
$$
```

```
LANGUAGE plpgsql
```

If you have an existing name for a variable, it is possible to change it with aliasing in the function declarations. We can map the variable i.e. \$1 is the first input variable and we can alias it to x.

30.7 Declaring Variables in a Function

```
CREATE OR REPLACE FUNCTION fn_my_sum(integer, integer) RETURNS integer AS  
$BODY$  
    DECLARE  
        ret integer;  
    BEGIN  
        ret := $1 + $2;  
        RETURN ret;  
    END;  
$BODY$  
LANGUAGE plpgsql;
```

```
SELECT fn_my_sum(2,2);
```

	fn_my_sum integer	
1		4

```
CREATE OR REPLACE FUNCTION fn_my_sum1(integer, integer) RETURNS integer AS
$BODY$
    DECLARE
        ret      integer;
        x ALIAS FOR $1;
        y ALIAS FOR $2;
    BEGIN
        ret := x + y;
        RETURN ret;
    END;
$BODY$
LANGUAGE plpgsql;
```

```
SELECT fn_my_sum1(2,2);
```

	fn_my_sum1	
	integer	
1		4

The advantage of declaring a variable using alias for is that it improves readability and I can now use x and y variables anywhere in my block section.

30.8 Variables Initializing Timing

```
DO
$$
    DECLARE
        start_time time:= NOW();
    BEGIN
        RAISE NOTICE 'Starting at %s: ',start_time;
        PERFORM pg_sleep(2);
        RAISE NOTICE 'Next time %s: ',start_time;

    END;
$$
LANGUAGE plpgsql;
```

```
NOTICE: Starting at 20:42:41.955478s:
NOTICE: Next time 20:42:41.955478s:
DO
```

```
Query returned successfully in 2 secs
42 msec.
```

30.9 Copying Data Types

%TYPE	refers to the type of table column or another variable.	DO \$\$ DECLARE
Variable_name		empl_first_name employees.first_name%TYPE; prod_name products.product_name%TYPE;
table_name.column_name%type;		BEGIN
i.e. declare a variable coming from database table.		... END; \$\$

30.10 Assigning Variables From Query

SELECT expression INTO variable_name Must return only a single result

SELECT * FROM products INTO product_row LIMIT 1; All the columns would be assigned to product_row variable.

SELECT product_row.product_name INTO product_name;

I can take the above variable and extract one column from the variable and assign it to another variable.

```

DO
$$
DECLARE
    product_title products.product_name%TYPE;
BEGIN
SELECT product_name
FROM products
INTO product_title
WHERE product_id = 1
LIMIT 1;
RAISE NOTICE 'Your product name is %', product_title;
END;
$$

```

```

NOTICE: Your product name is Chai
DO
Query returned successfully in 37 msec.

```

Return a specific column of a row.

```

DO
$$
DECLARE
    row_product record;
BEGIN
SELECT *
FROM products
INTO row_product
WHERE product_id = 1
LIMIT 1;
RAISE NOTICE 'Your product name is %', row_product.product_name;
END;
$$

```

```

NOTICE: Your product name is Chai
DO
Query returned successfully in 36 msec.

```

How about accessing the full result set? We will use a variable with 'record' data type.

30.11 Using IN, OUT Without RETURNS

We use the RETURNS clause in the first row of the function definition, however we can use IN, OUT, INOUT parameter modes.

IN	variable_name	type
OUT	variable_name	type
INOUT	variable_name	type

```
CREATE OR REPLACE FUNCTION my_fun_sum_2_par(IN x integer, IN y integer, OUT z integer) AS
$$
BEGIN
z := x + y;
END;
$$
LANGUAGE plpgsql;
```

```
SELECT my_fun_sum_2_par(2,2);
```

my_fun_sum_2_par	
integer	
1	4

Instead of specifying a RETURNS clause in the function creation line, I specify the input parameters with IN and the output parameters with OUT. But can I specify more than one output?

```
CREATE OR REPLACE FUNCTION my_fun_sum_1_par(IN x integer, IN y integer, OUT w integer, OUT z integer) AS
$$
BEGIN
z := x + y;
w := x * y;
END;
$$
LANGUAGE plpgsql;
```

```
SELECT my_fun_sum_1_par(3,2);
```

my_fun_sum_1_par	
record	
1	6,5

30.12 Variables in Block and Sub-Block

Block 1's v1 variable only belongs to block 1. Block 2's v2 variable only belongs to block 2.

If I need to reference a variable outside of the block then I reference the block name then dot followed by the variable name.

```
block_1
  DECLARE
    v1 int;
  BEGIN
    block_2
      DECLARE
        v2 int;
      BEGIN
        ...
      END;
    END;
```

```
DO
$$
  <<PARENT>>
  DECLARE
  counter integer := 0;
  BEGIN
    counter := counter +1;
    RAISE NOTICE 'THE CURRENT VALUE OF COUNTER IS %', counter;
    -- another block
    <<CHILD>>
    DECLARE counter integer := 0;
    BEGIN
      counter := counter +5;
      RAISE NOTICE 'THE CURRENT VALUE OF COUNTER AT CHILD BLOCK IS %', counter;
      RAISE NOTICE 'THE CURRENT VALUE OF COUNTER AT PARENT BLOCK IS %', parent.counter;
    END;
  END;
$$
```

```
NOTICE: THE CURRENT VALUE OF COUNTER IS 1
NOTICE: THE CURRENT VALUE OF COUNTER AT CHILD BLOCK IS 5
NOTICE: THE CURRENT VALUE OF COUNTER AT PARENT BLOCK IS 1
DO
```

```
Query returned successfully in 40 msec.
```


30.13 How to Return Query Results

```
CREATE OR REPLACE FUNCTION function_name RETURNS SETOF table_name AS
$$
BEGIN
RETURN QUERY SELECT...
END;
$$
LANGUAGE plpgsql
```

The syntax is similar to an SQL function where I use SETOF and table_name to return a set of results as a table.

Use **RETURN QUERY** in the body of the function.

```
CREATE OR REPLACE FUNCTION fn_api_orders_latest_top_10_orders() RETURNS SETOF orders AS
$$
BEGIN
RETURN QUERY
SELECT * FROM orders ORDER BY order_date DESC LIMIT 10;
END;
$$
LANGUAGE plpgsql;
```

```
SELECT * FROM fn_api_orders_latest_top_10_orders();
```

Showing rows: 1 to 10 Page No: 1 of 1									
	order_id smallint	customer_id character	employee_id smallint	order_date date	required_date date	shipped_date date	ship_via smallint	freight real	ship_name character varying (40)
1	11074	SIMOB	7	1998-05-06	1998-06-03	[null]	2	18.44	Simons bistro
2	11075	RICSU	8	1998-05-06	1998-06-03	[null]	2	6.19	Richter Supermarkt
3	11077	RATTC	1	1998-05-06	1998-06-03	[null]	2	8.53	Rattlesnake Canyon Grocery
4	11076	BONAP	4	1998-05-06	1998-06-03	[null]	2	28.28	Bon app'

```
CREATE OR REPLACE FUNCTION fn_api_orders_latest_top_10_orders() RETURNS SETOF orders AS
$$
BEGIN
RETURN QUERY
SELECT order_id FROM orders ORDER BY order_date DESC LIMIT 10;
END;
$$
LANGUAGE plpgsql;
```

```
SELECT * FROM fn_api_orders_latest_top_10_orders();
ERROR: Number of returned columns (1) does not match expected column count
(14).structure of query does not match function result type

ERROR: structure of query does not match function result type
SQL state: 42804
Detail: Number of returned columns (1) does not match expected column count (14).
Context: SQL statement "SELECT order_id FROM orders ORDER BY order_date DESC LIMIT
10" PL/pgSQL function fn_api_orders_latest_top_10_orders() line 3 at RETURN QUERY
```

When using SETOF and specifying the table name I pretty much want to use star to select all columns or it will throw an error because the columns in the body select query do not match the expected number of columns. i.e. 14 columns in the case of the orders table.

30.14 Control Structures – IF statements

Three Control Structures provided
by PL/pgSQL language →

- Conditional statements
 - IF
- Loop Statements
- Exception Handler Statements
 - CASE

IF
...
END IF;

Only two possible
conditions

IF Boolean-expression THEN
statement ...
END IF;

IF THEN
IF THEN ELSE
IF THEN ELSEIF
...
END IF;

Multiple possible
conditions

IF expression THEN
statement ...
ELSEIF expression THEN
statement ...
ELSEIF expression THEN
statement ...
ELSE
statement ...
END IF;

```
CREATE OR REPLACE FUNCTION fn_my_check(x integer DEFAULT 0, y integer DEFAULT 0) RETURNS
text AS
$$
BEGIN
IF x > y THEN
    RETURN 'x > y';
END IF;
END;
$$
LANGUAGE plpgsql;
```

30.15 Control Structures – IF statements

```
SELECT fn_my_check(1,2);  
ERROR: control reached end of function without RETURN  
CONTEXT: PL/pgSQL function fn_my_check(integer,integer)  
SQL state: 2F005
```

1 is not greater than 2 so the condition has not been met. My function generates this error because there is no control to handle what happens when the condition is not met.

```
CREATE OR REPLACE FUNCTION fn_my_check(x integer DEFAULT 0, y integer DEFAULT 0) RETURNS  
text AS  
$$  
BEGIN  
IF x > y THEN  
    RETURN 'x > y';  
ELSE  
    RETURN 'X < y';  
END IF;  
END;  
$$  
LANGUAGE plpgsql;
```

```
SELECT fn_my_check(1,2);
```

	fn_my_check text	
1	X < y	

```
SELECT fn_my_check(2,2);
```

	fn_my_check text	
1	X < y	

This is a subtle error. I am asking if 2 is greater than 2 which it is not so I am saying else return 'x < y' which is not true. So I need a third condition to handle if the two numbers are the same.

```
CREATE OR REPLACE FUNCTION fn_my_check(x integer DEFAULT 0, y integer DEFAULT 0) RETURNS
text AS
$$
BEGIN
IF x > y THEN
    RETURN 'x > y';
ELSEIF x = y THEN
    RETURN 'x = y';
ELSE
    RETURN 'x < y';
END IF;
END;
$$
LANGUAGE plpgsql;
```

```
SELECT fn_my_check(2,2);
```

	fn_my_check	
	text	
1	x = y	

30.16 Using IF with Table Data

```
CREATE OR REPLACE FUNCTION fm_api_products_catagory(price real) RETURNS text AS
$$
BEGIN
IF price > 50 THEN
    RETURN 'HIGH';
ELSEIF price > 25 THEN
    RETURN 'MEDUIM';
ELSE
    RETURN 'SWEET SPOT';
END IF;
END;
$$
LANGUAGE plpgsql;
```

```
SELECT fm_api_products_catagory(unit_price),* FROM products ORDER BY 1 DESC;
```

	fm_api_products_catagory text	product_id [PK] smallint	product_name character varying (40)	supplier_id smallint	category_id smallint	quantity_per_unit character varying (20)	unit_price real	u s
1	SWEET SPOT	77	Original Frankfurter grüne Soße	12	2	12 boxes	13	
2	SWEET SPOT	2	Chang	1	1	24 - 12 oz bottles	19	
3	SWEET SPOT	3	Aniseed Syrup	1	2	12 - 550 ml bottles	10	
4	SWEET SPOT	4	Chef Anton's Cajun Seasoning	2	2	48 - 6 oz jars	22	
5	SWEET SPOT	5	Chef Anton's Gumbo Mix	2	2	36 boxes	21.35	
6	SWEET SPOT	6	Grandma's Boysenberry Spread	3	2	12 - 8 oz jars	25	
7	SWEET SPOT	36	Inlagd Sill	17	8	24 - 250 g jars	19	
8	SWEET SPOT	39	Chartreuse verte	18	1	750 cc per bottle	18	
9	SWFFT SPOT	40	Boston Crab Meat	19	8	24 - 4 oz tins	18.4	

30.17 CASE Statement

Simple if we have to make a choice from a LIST of values.
Searched we have to chose from a RANGE of values.

```
CASE simple_expression
  WHEN condition
    THEN statement
  WHEN condition
    THEN statement
  ELSE statement
END CASE;
```

```
CREATE OR REPLACE FUNCTION fn_my_check_value(x integer DEFAULT 0) RETURNS text AS
$$
BEGIN
CASE x
  WHEN 10 THEN
    RETURN 'VALUE IS 10';
  WHEN 20 THEN
    RETURN 'VALUE IS 20';
  ELSE
    RETURN 'No values found, returning input value x';
END CASE;
END;
$$
LANGUAGE plpgsql;
```

```
SELECT fn_my_check_value(10);
```

fn_my_check_value	
text	
1	VALUE IS 10

```
SELECT fn_my_check_value(20);
```

fn_my_check_value	
text	
1	VALUE IS 20

```
SELECT fn_my_check_value(1);
```

fn_my_check_value	
text	
1	No values found, returning input value x

```

CREATE OR REPLACE FUNCTION fn_api_ordder_ship_via(ship_via SMALLINT) RETURNS text
AS
$$
BEGIN
CASE ship_via
    WHEN 1 THEN
        RETURN 'Speedy Express';
    WHEN 2 THEN
        RETURN 'United Package';
    WHEN 3 THEN
        RETURN 'Federal Shipping';
    WHEN 4 THEN
        RETURN 'Alliance Shippers';
    ELSE
        RETURN 'UPS or DHL';
END CASE;
END;
$$
LANGUAGE plpgsql;

```

```

SELECT fn_api_ordder_ship_via(ship_via),* FROM orders;

```

	fn_api_ordder_ship_via  text	order_id  [PK] smallint	customer_id  character	employee_id  smallint	order_date  date	required_date  date	shipped_date  date	ship_via  smallint	freight  real
1	Federal Shipping	10248	VINET	5	1996-07-04	1996-08-01	1996-07-16	3	32.3
2	Speedy Express	10249	TOMSP	6	1996-07-05	1996-08-16	1996-07-10	1	11.6
3	United Package	10250	HANAR	4	1996-07-08	1996-08-05	1996-07-12	2	65.8
4	Speedy Express	10251	VICTE	3	1996-07-08	1996-08-05	1996-07-15	1	41.3
5	United Package	10252	SUPRD	4	1996-07-09	1996-08-06	1996-07-11	2	51.
6	United Package	10253	HANAR	3	1996-07-10	1996-07-24	1996-07-16	2	58.1

30.17 Searched CASE Statement

```
DO
$$
DECLARE
    total_amount numeric;
    order_type varchar(50);
BEGIN
    SELECT
        SUM((unit_price * quantity) - discount) INTO total_amount
        FROM order_details
        WHERE order_id = 10248;

    IF found THEN
        CASE
            WHEN total_amount > 200
            THEN order_type = 'PLATINUM';
            WHEN total_amount > 100
            THEN order_type = 'GOLD';
            ELSE
                order_type = 'SILVER';
            END CASE;
        RAISE NOTICE 'Order Amount, Order Type % %', total_amount, order_type;
    ELSE
        RAISE NOTICE 'No order was found.';
    END IF;
END;
$$
LANGUAGE plpgsql;
```

Note that because I am checking for a range i.e. `total_amount > 200` (0...200) then I should not specify the variable after declaring CASE

```
NOTICE: Order Amount, Order Type 439.999998092651 PLATINUM
DO
```

```
Query returned successfully in 52 msec.
```

```

DO
$$
DECLARE
    total_amount numeric;
    order_type varchar(50);
BEGIN
    SELECT
        SUM((unit_price * quantity) - discount) INTO total_amount
        FROM order_details
    WHERE order_id = 10248;

    IF found THEN
        CASE total_amount
            WHEN total_amount > 200
                THEN order_type = 'PLATINUM';
            WHEN total_amount > 100
                THEN order_type = 'GOLD';
            ELSE
                order_type = 'SILVER';
        END CASE;
        RAISE NOTICE 'Order Amount, Order Type % %', total_amount, order_type;
    ELSE
        RAISE NOTICE 'No order was found.';
    END IF;
END;
$$
LANGUAGE plpgsql;

```

```

ERROR: operator does not exist: numeric = boolean
LINE 1: "__Case__Variable_4__" IN (total_amount > 200)
                                     ^
HINT: No operator matches the given name and argument types. You
might need to add explicit type casts.
QUERY: "__Case__Variable_4__" IN (total_amount > 200)
CONTEXT: PL/pgSQL function inline_code_block line 12 at CASE

SQL state: 42883

```

30.18 LOOP Statement

The LOOP statement allows us to execute a block of code endlessly so it is essential to include an exit statement.

```
LOOP
    statement
EXIT;
END LOOP;
```

```
LOOP
    statement
EXIT WHEN condition (met);
END LOOP;
```

```
LOOP
    statement
IF condition THEN
    EXIT;
END LOOP;
```

```
DO
$$
DECLARE
    i_counter integer := 0;
BEGIN
    LOOP
        RAISE NOTICE '%', i_counter;
        i_counter := i_counter + 1;
    EXIT WHEN
        i_counter := 5;
    END LOOP
END;
$$
LANGUAGE plpgsql;
```

```
NOTICE:  0
NOTICE:  1
NOTICE:  2
NOTICE:  3
NOTICE:  4
DO
```

```
Query returned successfully in 44 msec.
```

30.19 FOR Loops

The FOR loop can be used to loop through a range of integers, objects or data coming up from the table. Note the double dot between the start value and end value.

```
FOR [counter name] IN [REVERSE] [START VALUE] .. [END VALUE] [BY stepping]
LOOP
    [statements];
END LOOP;
```

```
DO
$$
BEGIN
    FOR counter IN 1..10
    LOOP
        RAISE NOTICE 'counter : %',
counter;
    END LOOP;
END;
$$
LANGUAGE plpgsql;
```

```
NOTICE: counter : 1
NOTICE: counter : 2
NOTICE: counter : 3
NOTICE: counter : 4
NOTICE: counter : 5
NOTICE: counter : 6
NOTICE: counter : 7
NOTICE: counter : 8
NOTICE: counter : 9
NOTICE: counter : 10
DO
```

Query returned successfully in 62 msec.

```
$$
BEGIN
    FOR counter IN REVERSE 10..5
    LOOP
        RAISE NOTICE 'counter : %',
counter;
    END LOOP;
END;
$$
LANGUAGE plpgsql;
```

```
NOTICE: counter : 10
NOTICE: counter : 9
NOTICE: counter : 8
NOTICE: counter : 7
NOTICE: counter : 6
NOTICE: counter : 5
DO
```

Query returned successfully in 37 msec.

```

$$
BEGIN
    FOR counter IN REVERSE 10..1 BY 2
    LOOP
        RAISE NOTICE 'counter : %', counter;
    END LOOP;
END;
$$
LANGUAGE plpgsql;

```

```

NOTICE: counter : 1
NOTICE: counter : 3
NOTICE: counter : 5
NOTICE: counter : 7
NOTICE: counter : 9
DO

```

Query returned successfully in 37 msec.

The use of the BY keyword means that I can specify a value by which is steps or skips. i.e. it starts at 1 and skips to the 3rd, then 5th then 7th then 9th value.

30.20 FOR Loops Iterate over Result Set

```

FOR target IN your_select_statement
LOOP

```

```

END LOOP;

```

```

DO
$$
DECLARE
    rec record;
BEGIN
    FOR rec IN
        SELECT order_id, customer_id FROM orders LIMIT 10
    LOOP
        RAISE NOTICE '% %', order_id, customer_id;
    END LOOP;
END;
$$
LANGUAGE plpgsql;

```

```

ERROR: column "order_id" does not exist
LINE 1: order_id
          ^
QUERY: order_id
CONTEXT: PL/pgSQL function inline_code_block line 8 at RAISE
SQL state: 42703

```

The error is due to the whole select query being passed to rec.

```
DO
$$
DECLARE
    rec record;
BEGIN
    FOR rec IN
        SELECT order_id, customer_id FROM orders LIMIT 10
    LOOP
        RAISE NOTICE '% %', rec.order_id, rec.customer_id;
    END LOOP;
END;
$$
LANGUAGE plpgsql;
```

```
NOTICE: 10248 VINET
NOTICE: 10249 TOMSP
NOTICE: 10250 HANAR
NOTICE: 10251 VICTE
NOTICE: 10252 SUPRD
NOTICE: 10253 HANAR
NOTICE: 10254 CHOPS
NOTICE: 10255 RICSU
NOTICE: 10256 WELLI
NOTICE: 10257 HILAA
DO
```

Query returned successfully in 45 msec.

30.21 CONTINUE Statement

CONTINUE [**loop_label**] [**when condition**]

The continue statement skips the current iteration of the loop and jumps to the next one.

If the loop label is specified then it skips the current of that loop.

```
DO
$$
DECLARE
    i_counter INTEGER =0;
BEGIN
    LOOP
        i_counter = i_counter + 1;
        EXIT WHEN i_counter > 20;
        CONTINUE WHEN MOD(i_counter,2) = 1;
        RAISE NOTICE '%', i_counter;
    END LOOP;
END;
$$
LANGUAGE plpgsql;
```

```
NOTICE:  2
NOTICE:  4
NOTICE:  6
NOTICE:  8
NOTICE: 10
NOTICE: 12
NOTICE: 14
NOTICE: 16
NOTICE: 18
NOTICE: 20
DO
```

```
Query returned successfully in 49 msec.
```

30.21 FOREACH Loop with Arrays

FOREACH var IN ARRAY array_name

```
DO
$$
DECLARE
    arr1 INT[] :=array[1,2];
    var int;
BEGIN
    FOREACH var IN ARRAY arr1
    LOOP
        RAISE INFO '%', var;
    END LOOP;
END;
$$
LANGUAGE plpgsql;
```

```
INFO: 1
INFO: 2
DO
```

Query returned successfully in 39 msec.

I can declare arr1 as an array of integers and set the default values in the array to 1 and 2.

In the foreach loop I iterate over the array assigning each value the the variable var.

```
DO
$$
DECLARE
    arr1 INT[] :=array[1,2,3,4];
    arr2 INT[] :=array[5,6,7,8];
    var int;
BEGIN
    FOREACH var IN ARRAY arr1||arr2
    LOOP
        RAISE INFO '%', var;
    END LOOP;
END;
$$
LANGUAGE plpgsql;
```

```
INFO: 1
INFO: 2
INFO: 3
INFO: 4
INFO: 5
INFO: 6
INFO: 7
INFO: 8
DO
```

Query returned successfully in 41 msec.

The double pipe || is like an array merge.

30.22 WHILE Loop

```
WHILE your_condition  
LOOP  
    statements ...  
END LOOP;
```

A while loop evaluates the condition defined to determine if the loop should be continued. If the condition evaluates to true the loop is continued.

```
CREATE OR REPLACE FUNCTION fn_while_loop_sum_all(x integer) RETURNS numeric AS  
$$  
DECLARE  
    counter integer := 1;  
    sum_all integer := 0;  
BEGIN  
    WHILE counter <= x  
    LOOP  
        sum_all := sum_all + counter;  
        counter := counter + 1;  
    END LOOP;  
    RETURN sum_all;  
  
END;  
$$  
LANGUAGE plpgsql;
```

```
SELECT fn_while_loop_sum_all(4);
```

1+2+3+4 = 10

fn_while_loop_sum_all numeric	
1	10

```
SELECT fn_while_loop_sum_all(9);
```

1+2+3+4+5+6+7+8+9 = 45

fn_while_loop_sum_all numeric	
1	45

```

CREATE OR REPLACE FUNCTION fn_create_t_table_insert_values(x integer) RETURNS boolean AS
$$
DECLARE
    counter integer :=1;
    done boolean :=false;
BEGIN
    EXECUTE format('CREATE TABLE IF NOT EXISTS t_table (id int)');
    WHILE counter <= x
    LOOP
        INSERT INTO t_table (id) VALUES (counter);
        counter :=counter + 1;
    END LOOP;
    RETURN done;
END;
$$
LANGUAGE plpgsql;

```

```
SELECT fn_create_t_table_insert_values(10);
```

	fn_create_t_table_insert_values 
1	false

```
SELECT * FROM t_table;
```

	id 
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	10

This while loop uses a counter to count up to a value of x and add the value as a row in a table.

The create table part is outside of the while loop because I only want to create the table once.

30.23 Using RETURN QUERY

```
CREATE OR REPLACE FUNCTION fn_api_products_sold_more_than_val(p_total_sales real) RETURNS
SETOF products AS
$$
BEGIN
    RETURN QUERY
    SELECT * FROM products WHERE product_id IN
    (
        SELECT product_id FROM
        (
            SELECT
            product_id,
            SUM((unit_price * quantity) - discount)
            FROM order_details
            GROUP BY product_id
            HAVING SUM((unit_price * quantity) - discount) > p_total_sales
        ) AS filter_products
    );
    IF NOT FOUND THEN
        RAISE EXCEPTION 'No products were found for total sales > %', p_total_sales;
    END IF;
END;
$$
LANGUAGE plpgsql;
```

```
SELECT * FROM fn_api_products_sold_more_than_val(10000);
```

Showing rows: 1 to 40 Page No: 1 of 1									
	product_id smallint	product_name character varying (40)	supplier_id smallint	category_id smallint	quantity_per_unit character varying (20)	unit_price real	units_in_stock smallint	units_on_order smallint	
1	1	Chai	8	1	10 boxes x 30 bags	18	39	0	
2	2	Chang	1	1	24 - 12 oz bottles	19	17	40	
3	7	Uncle Bob's Organic Dried Pears	3	7	12 - 1 lb pkgs.	30	15	0	

```
SELECT * FROM fn_api_products_sold_more_than_val(500000);  
ERROR: No products were found for total sales > 500000 CONTEXT: PL/pgSQL  
function fn_api_products_sold_more_than_val(real) line 17 at RAISE  
SQL state: P0001
```

Testing to see that the exception works

30.24 Returning a Table

```
CREATE OR REPLACE FUNCTION fn_api_get_products_by_names(p_pattern VARCHAR)  
RETURNS TABLE (productname VARCHAR, unitprice REAL) AS  
$$  
BEGIN  
RETURN QUERY  
SELECT  
product_name,  
unit_price,  
FROM products  
WHERE product_name LIKE p_pattern;  
END;  
$$  
LANGUAGE plpgsql;
```

```
SELECT * FROM fn_api_get_products_by_names('B%');
```

	productname character varying	unitprice real
1	Boston Crab Meat	18.4

Note that using a p_pattern input is vulnerable to SQL injection. This exercise is purely to demonstrate how to create a function that would output a table using RETURNS TABLE (column_list)

30.25 using RETURN NEXT

```
CREATE OR REPLACE FUNCTION fn_get_all_orders_greater_than() RETURNS SETOF order_details
AS
$$
DECLARE
r record;
BEGIN
    FOR r IN
        SELECT * FROM order_details WHERE unit_price > 100
    LOOP
        RETURN NEXT r;
    END LOOP;
    RETURN;
END;
$$
LANGUAGE plpgsql;
```

```
SELECT * FROM fn_get_all_orders_greater_than();
```

	order_id smallint	product_id smallint	unit_price real	quantity smallint	discount real
1	10329	38	210.8	20	0.05
2	10351	38	210.8	20	0.05
3	10353	38	210.8	50	0.2
4	10360	38	210.8	10	0
5	10372	38	210.8	40	0.25
6	10417	38	210.8	50	0
7	10424	38	210.8	40	0.2

```

CREATE OR REPLACE FUNCTION fn_products_new_dec_pricing() RETURNS SETOF products AS
$$
DECLARE
r record;
BEGIN
    FOR r IN
        SELECT * FROM products
    LOOP
        CASE
            WHEN r.category_id IN (1,2,3) THEN
                r.unit_price = r.unit_price * 0.90;
            WHEN r.category_id IN (4,5,6) THEN
                r.unit_price = r.unit_price * 0.80;
            ELSE
                r.unit_price = r.unit_price * 1.20;
            END CASE;
        RETURN NEXT r;
    END LOOP;
    RETURN;
END;
$$
LANGUAGE plpgsql;

```

```
SELECT * FROM fn_products_new_dec_pricing();
```

	product_id smallint	product_name character varying (40)	supplier_id smallint	category_id smallint	quantity_per_unit character varying (20)	unit_price real	u s
1	1	Chai	8	1	10 boxes x 30 bags	16.2	
2	2	Chang	1	1	24 - 12 oz bottles	17.1	
3	3	Aniseed Syrup	1	2	12 - 550 ml bottles	9	
4	4	Chef Anton's Cajun Seasoning	2	2	48 - 6 oz jars	19.8	
5	5	Chef Anton's Gumbo Mix	2	2	36 boxes	19.215	

30.26 Error and Exception Handling

When creating a function it is important to also handle the errors whether they are errors with how the function is written or if there is output that is erroneous from the SQL, too many rows, etc. The exception clause stores all the errors.

```
DO
$$
DECLARE
    rec record;
    orderid smallint = 1;
BEGIN
    SELECT
        customer_id,
        order_date
    FROM orders
    INTO STRICT rec
    WHERE order_id = orderid;
END;
$$
LANGUAGE plpgsql;
```

```
ERROR: query returned no rows
CONTEXT: PL/pgSQL function inline_code_block line 6 at SQL statement

SQL state: P0002
```

This error is expected because there is no order with an id of 1.

```
DO
$$
DECLARE
    rec record;
    orderid smallint = 1;
BEGIN
    SELECT
        customer_id,
        order_date
    FROM orders
    INTO STRICT rec
    WHERE order_id = orderid;
    EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RAISE EXCEPTION 'No order Id was found %', orderid;
END;
$$
LANGUAGE plpgsql;
```

```
ERROR: No order Id was found 1
CONTEXT: PL/pgSQL function inline_code_block line 14 at RAISE

SQL state: P0001
```

30.27 EXCEPTION – Too many Rows

```
DO
$$
DECLARE
    rec record;
BEGIN
    SELECT
        customer_id,
        company_name
    FROM customers
    INTO STRICT rec
    WHERE company_name LIKE 'A%';
END;
$$
LANGUAGE plpgsql;
```

```
ERROR: query returned more than one row
HINT: Make sure the query returns a single row, or use LIMIT 1.
CONTEXT: PL/pgSQL function inline_code_block line 5 at SQL
statement
```

```
SQL state: P0003
```

This code returned too_many_rows so we can use it to build a test exception.

```
DO
$$
DECLARE
    rec record;
BEGIN
    SELECT
        customer_id,
        company_name
    FROM customers
    INTO STRICT rec
    WHERE company_name LIKE 'A%';
    EXCEPTION
    WHEN too_many_rows THEN
        RAISE EXCEPTION 'Your query returns too many rows';
END;
$$
LANGUAGE plpgsql;
```

```
ERROR: Your query returns too many rows
CONTEXT: PL/pgSQL function inline_code_block line 13 at RAISE
SQL state: P0001
```

The exception handling code returns SQL state: P0001 for too_many_rows

30.28 Using SQL STATE codes for exception Handling

All messages emitted by the PostgreSQL server are assigned five-character error codes that follow the SQL standard's conventions for “SQLSTATE” codes. Applications that need to know which error condition has occurred should usually test the error code, rather than looking at the textual error message.

<https://www.postgresql.org/docs/current/errcodes-appendix.html>

Class P0 are the error codes for PL/pgSQL.

Class P0 — PL/pgSQL Error	
P0000	plpgsql_error
P0001	raise_exception
P0002	no_data_found
P0003	too_many_rows
P0004	assert_failure

```
DO
$$
DECLARE
    rec record;
    orderid smallint = 1;
BEGIN
    SELECT
        customer_id,
        order_date
    FROM orders
    INTO STRICT rec
    WHERE order_id = orderid;
    EXCEPTION
    WHEN SQLSTATE 'P0002' THEN
        RAISE EXCEPTION 'No orderid was found %', orderid;
END;
$$
LANGUAGE plpgsql;
```

```
ERROR: No orderid was found 1
CONTEXT: PL/pgSQL function inline_code_block line 14 at RAISE
SQL state: P0001
```

I can use the specific error codes in the exception handling statement.

In this case I am intercepting the error code for no_data_found (P0002) and building a custom error message.

30.29 Exception with Data Exception Errors

Class 22 — Data Exception

22000	data_exception
2202E	array_subscript_error
22021	character_not_in_repertoire
22008	datetime_field_overflow
22012	division_by_zero
22005	error_in_assignment
2200R	escape character conflict

In this simulation I am going to use the division_by_zero 22012 error from the class 22 – Data exception errors.

<https://www.postgresql.org/docs/current/errcodes-appendix.html>

This simple function is to divide x by y. if I call the function and try and divide a number by zero then it will throw a 22012 error.

```
CREATE OR REPLACE FUNCTION fn_div_exception(x real, y real)
RETURNS real AS
$$
DECLARE
    ret real;
BEGIN
    ret := x / y;
    return ret;
END;
$$
LANGUAGE plpgsql;
```

```
SELECT fn_div_exception(5,0);
ERROR: division by zero
CONTEXT: PL/pgSQL function fn_div_exception(real,real)
line 5 at assignment
```

```
SQL state: 22012
```

```

CREATE OR REPLACE FUNCTION fn_div_exception(x real, y real)
RETURNS real AS
$$
DECLARE
    ret real;
BEGIN
    ret := x / y;
    return ret;
EXCEPTION
    WHEN SQLSTATE '22012' THEN
        RAISE EXCEPTION '% cannot be divided by %', x, y;
END;
$$
LANGUAGE plpgsql;

```

```

SELECT fn_div_exception(5,0);
ERROR: 5 cannot be divided by 0
CONTEXT: PL/pgSQL function fn_div_exception(real,real)
line 9 at RAISE

SQL state: P0001

```

I can add an exception code to intercept and handle the 22012 error.

```

CREATE OR REPLACE FUNCTION fn_div_exception(x real, y real)
RETURNS real AS
$$
DECLARE
    ret real;
BEGIN
    ret := x / y;
    return ret;

EXCEPTION
    WHEN division_by_zero THEN
        RAISE INFO 'DIVISION by zero';
        RAISE INFO 'Error % %', SQLSTATE, SQLERRM;
END;
$$
LANGUAGE plpgsql;

```

```

SELECT fn_div_exception(5,0);
INFO: DIVISION by zero
INFO: Error 22012 division by zero

ERROR: control reached end of function without RETURN
CONTEXT: PL/pgSQL function fn_div_exception(real,real)

SQL state: 2F005

```

Or I can use the text name of the error code

```

CREATE OR REPLACE FUNCTION fn_div_exception(x real, y real)
RETURNS real AS
$$
DECLARE
    ret real;
BEGIN
    ret := x / y;
    return ret;
EXCEPTION
    WHEN division_by_zero THEN
        RAISE INFO 'DIVISION by zero';
        RAISE INFO 'Error % %', SQLSTATE, SQLERRM;
        RETURN 0;
END;
$$
LANGUAGE plpgsql;

```

```
SELECT fn_div_exception(5,0);
```

	fn_div_exception real
1	0

Instead of allowing the error to return an incomplete **ERROR: control reached end of function without RETURN**, I can add a return zero statement o the end of the exception handling.

I can use this technique for handling data exception errors.

31. Exploring Stored Procedures

[31.1 Functions vs Stored Procedures](#)

[31.2 Create Transactions](#)

[31.3 Understanding the Use of Stored Procedures](#)

[31.4 Returning a Value](#)

[31.5 Drop a Procedure](#)

31.01 Functions vs Stored Procedures

1. User-defined functions (CREATE FUNCTION ...) cannot execute transactions. i.e. inside a function you cannot start a transaction.
2. Stored procedures support transactions.
3. Stored procedures do not return values so you cannot use 'return'. However you can use the return statement without the expression to stop the stored procedure immediately. INOUT mode can be used to return a value from a stored procedure.
4. Can not be executed or called from within a SELECT.
5. You can call a procedure as often as you like.
6. They are a compiled object
7. Procedures may or may not use parameters (argument values)
8. Execution
 - Implicit execution, EXECUTE command, along with specific SP name and optional parameters
 - Implicit execution using only SP name. CALL procedure_name()
9. Declaration section can be empty.
10. Stored procedures that do not have parameters (arguments) are called "static".
11. Stored procedures that use parameter values are called "dynamic".

31.2 Create Transactions

```
CREATE TABLE t_accounts(  
  recid SERIAL PRIMARY KEY,  
  name VARCHAR NOT NULL,  
  balance DEC(15,2) NOT NULL  
);
```

```
INSERT INTO t_accounts (name, balance) VALUES  
( 'Adam', 100),  
( 'Linda', 100);
```

```
SELECT * FROM t_accounts;
```

	recid [PK] integer	name character varying	balance numeric (15,2)
1	1	Adam	100.00
2	2	Linda	100.00

```
CREATE OR REPLACE PROCEDURE pr_money_transfer(  
  sender int,  
  receiver int,  
  amount decimal  
)  
AS  
$$  
BEGIN  
  -- subtract ammount from sender account  
  UPDATE t_accounts SET balance = balance - amount  
  WHERE recid = sender;  
  -- add amount to receiver account  
  UPDATE t_accounts SET balance = balance + amount  
  WHERE recid = receiver;  
  
  COMMIT;  
END;  
$$  
LANGUAGE plpgsql;
```

This procedure will perform a simple money transfer between two accounts. However it has some design flaws. If simultaneously there is another transaction occurring at the same time then the sender account may not have the funds to send by the time this transaction executes.

```
CALL pr_money_transfer(1,2,100);  
CALL  
  
Query returned successfully in 55 msec.
```

```
SELECT * FROM t_accounts;
```

	recid [PK] integer	name character varying	balance numeric (15,2)
1	1	Adam	0.00
2	2	Linda	200.00

The transfer was performed maintaining data integrity.

31.3 Understanding the Use of Stored Procedures

1. To ensure data consistency by not requiring that a series of steps to be run or created over and over. Also if all the DBAs/Developers and even applications use the same stored procedures, then the SAME CODE will always be used.
2. To simplify complex operations and encapsulate that into a single easy-to-use unit.
3. To simplify overall 'change management' i.e. if tables, columns or business logic needs to be changed then only the stored procedures need to be updated instead of all changes at every level.
4. To ensure security, i.e. Restricting access to underlying data via stored procedures will reduce the chance of data corruption and more.
5. To fully utilise 'transaction' and it's benefits for data integrity and much more.
6. Performance. The code is compiled only when created, meaning no need to require at run-time, unless you change the program (stored Procedure).
7. Modularity. When you find yourself writing very similar queries multiple times in your code, instead you can chose to write one stored procedure that - maybe with a parameter or two – can be called over and over from your code, with far less typing, potential typos and future maintenance.
8. Security. Reduces potential for various kinds of hacking, including SQL query injection.

31.4 Returning a Value

```
CREATE OR REPLACE PROCEDURE pr_orders_count(INOUT total_count int DEFAULT 0) AS
$$
BEGIN
    SELECT COUNT(*) INTO total_count FROM ORDERS;
END;
$$
LANGUAGE plpgsql;
```

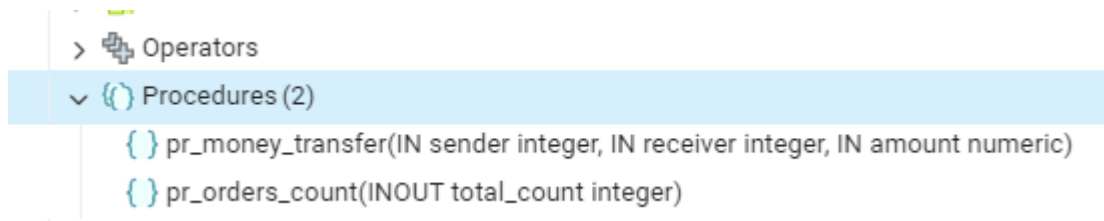
```
CALL pr_orders_count();
```

I can use the INOUT mode to return a value from a stored procedure.

	total_count integer
1	830

31.5 Drop a Procedure

```
DROP PROCEDURE [IF EXISTS] procedure_name (arguments_list) [CASCADE|RESTRICT];
```



The procedures show up in pgAdmin4 under database → schema → procedures

The drop procedure is like any other drop command and should be used with caution especially if using cascade. It is much safer to manually drop dependencies than performing a cascade drop.

32. PostgreSQL Triggers

[32.1 Introduction to Triggers](#)

[32.2 What is a Trigger?](#)

[32.3 Types of Trigger](#)

[32.4 Trigger Table](#)

[32.5 Pros & Cons of Triggers](#)

[32.6 Trigger Key Points](#)

[32.7 Steps to Create a Trigger](#)

[32.8 Data Auditing with Triggers](#)

[32.9 Bind a Function to a Table with a Trigger](#)

[32.10 Modify Data at INSERT event](#)

[32.11 View Trigger Variables](#)

[30.12 Disallow DELETE on a Table](#)

[30.13 Disallow TRUNCATE on a Table](#)

[30.14 Creating an Audit Trigger](#)

[30.15 Creating Conditional Triggers](#)

[30.16 Disallow data change on Primary Key](#)

[30.17 Use Triggers Very Cautiously](#)

[30.18 What is an Event Trigger](#)

[30.19 Event Triggers Usage Scenarios](#)

[30.20 Creating Event Triggers](#)

[30.21 Event Trigger Events & Variables](#)

[30.22 Creating and Audit Event Trigger](#)

[30.23 Prevent Schema Changes](#)

[30.24 Dropping a Trigger](#)

32.1 Introduction to Triggers

In the course of executing a business decision, or a business logic in a database application, special occasions may arise where if some specific action occurs:

You want the action to cause another action, or perhaps a succession of actions to occur.

In a way that first action, TRIGGERS the execution of the following actions. The SQL provides the TRIGGER mechanism to provide capability.

A trigger is an action or event that causes another event to occur.

A triggering SQL statement causes another SQL statement (the trigger statement) to be executed.

Triggers can also be used to keep a database consistent. For example, in an order entry application, an order for a specific product can trigger a statement that can change the status of that product in the master inventory table status to available to reserved. Similarly, the deletion of a row in the order table can trigger a statement that changes the status of the subject product status from reserved to available.

32.2 What is a Trigger?

1. A 'trigger' is defined as an event that sets a course of action in motion.
2. A PostgreSQL trigger is a function invoked automatically whenever 'an event' associated with a table occurs.
3. An event could be any of the following: INSERT, UPDATE, DELETE, TRUNCATE
4. A trigger can be associated with a specific: Table, view or foreign table
5. A trigger is a special 'user-defined function'
6. The difference between a trigger and a user-defined function is that a trigger is automatically invoked when a trigger event occurs. Comparable to a cron job.
7. We can create a trigger
 - BEFORE - If the trigger is invoked before an event, it can skip the operation for the current row or even change the row being updated or inserted.
 - AFTER - All changes are available to the trigger.
 - INSTEAD - of the events/operation.
8. If there are more than one triggers on a table, they are fired in an alphabetical order. First all of those BEFORE triggers happen in alphabetical order. Then PostgreSQL performs the row operation that the trigger has been fired for, and continues executing AFTER triggers in alphabetical order. In other words, the execution order of triggers are absolutely deterministic, and the number of triggers is basically unlimited.
9. Triggers can modify data before or after the actual modification has happened. In general, this is a good way to verify data and to error out if some custom restrictions are violated.
10. There are two main characteristics that make triggers different than stored procedures:
 - Triggers cannot be manually executed by the users.
 - There is no chance for triggers to receive parameters.

32.3 Types of Trigger

1. Row level triggers

If the trigger is marked FOR EACH ROW then the trigger function will be called for each row that is getting modified by the event.

e.g. if we update 20 rows in the table, the UPDATE trigger function will be called 20 times, once for each updated row.

2. Statement level trigger

The FOR EACH ROW option will call the trigger function only ONCE for each statement, regardless of the number of rows getting modified.

- As you can see, the difference between the two kinds are HOW MANY TIMES the trigger is invoked and AT WHAT TIME.
- In PostgreSQL, triggers have become more powerful over the years, and they now provide a rich set of features.

32.4 Trigger Table

The Following table will help to understand when to use which triggers in what event.

When	Event	Row-Level	Statement-Level
	INSERT / UPDATE / DELETE	Tables	Tables & Views
BEFORE	TRUNCATE	-	Tables
	INSERT / UPDATE / DELETE	Tables	Tables & Views
AFTER	TRUNCATE	-	Tables
	INSERT / UPDATE / DELETE	views	-
INSTEAD OF	TRUNCATE	-	-

32.5 Pros & Cons of Triggers

PRO's

1. Triggers are not difficult to code. The fact that they are coded like stored procedures makes getting started with triggers easy.
2. Triggers allow you to create 'basic auditing'. By using the delete table inside a trigger you can build a decent audit solution that inserts the contents of the deleted table data into an audit table which holds the data that is either being removed by a DELETE statement or being changed by an UPDATE statement.
3. You can call stored procedures and functions from inside a trigger.
4. Triggers are useful when you need to validate inserted or updated data into batches instead of row by row.
5. You can use triggers to implement referential integrity across databases. Unfortunately SQL server's don't allow the creation of constraints between objects on different databases, but by using triggers you can simulate the behaviour of constraints.
6. Triggers are useful if you need to be sure that certain events always happen when data is inserted, updated, or deleted. This is the case when you have to deal with complex default values on columns, or modify, the data of other tables.
7. Triggers can allow recursion. Triggers are recursive when a trigger on a table performs an action on the base table that causes another instance of the trigger to fire. This is useful when you have to solve a self-referencing relation, I.e. a constraint to itself.

CON's

1. Triggers are difficult to locate unless you have proper documentation because they are invisible to the client. Sometimes you execute DML statement without errors or warnings, say an insert, and you don't see it reflected in the table's data. In such a case you have to check the table for triggers that may be disallowing you to run the insert you wanted.
2. Triggers can add overhead to DML statements. Every time you run a delete modify statement that has a trigger associated to it, you are actually executing the DML statement and the trigger; but by definition the DML statement won't end until the trigger execution completes. This can create a problem in production.
3. The problem of using Triggers for audit purposes is that when triggers are enabled, they execute always regardless of the circumstances that caused the trigger to fire. For example, if you only need to audit the data inserted by a specific stored procedure and you use a trigger, you may have to delete the rows of the audit table that were created when someone changed data using an ad-hoc query or add more logic into the trigger's code which of course impacts performance. In such case you may need to use the OUTPUT clause.
4. If there are many nested triggers, it could get very hard to debug and troubleshoot, which consumes development time and resources.
5. Recursive triggers are even more harder to debug than the nested triggers.

32.6 Trigger Key points

1. No trigger on SELECT statement. Because SELECT DOES NOT MODIFY A ROW. In such cases lets use a view.
2. Multiple triggers can be used in alphabetical order.
3. UFD (user-defined functions) are allowed in triggers.
4. A single trigger can support MULTIPLE ACTIONS i.e. SINGLE → many.

32.7 Steps to Create a Trigger

To create a new trigger in PostGreSQL, follow these steps:

1. First, create a trigger function using CREATE FUNCTION statement
2. Second, bind the trigger function to a table by using CREATE TRIGGER statement.

```
CREATE FUNCTION trigger_function()  
    RETURNS TRIGGER  
    LANGUAGE plpgsql  
  
AS  
$$  
BEGIN  
    -- Trigger logic  
END;  
$$
```

```
CREATE TRIGGER trigger_name  
    {BEFORE | AFTER} {event }  
ON table_name  
    [FOR [EACH] {ROW | STATEMENT}]  
EXECUTE PROCEDURE trigger_function
```

- First specify the name of the trigger after the TRIGGER keywords
- Second, specify the timing that cause the trigger to fire. It can be BEFORE or AFTER the event occurs.
- Third, specify the event that invokes the trigger. The event can be INSERT, DELETE, UPDATE, or TRUNCATE.
- Fourth, specify the name of the table associated with the trigger after the ON keyword.
- Fifth, specify the type of triggers which can be:
 - Row-level trigger: That is specified by the FOR EACH ROW clause
 - Statement-level trigger: That is specified by the FOR EACH STATEMENT clause.
- Sixth, Execute the function.

32.8 Data Auditing with a Trigger

The lab scenario is that we have a 'players' table. Suppose we want to log all data when the name of a player gets changed or updated. We need a player_audits table to log all the changes.

```
CREATE TABLE players (  
    player_id SERIAL PRIMARY KEY,  
    name VARCHAR(100)  
);
```

```
CREATE TABLE players_audit(  
    player_audit_id SERIAL PRIMARY KEY,  
    player_id INT NOT NULL,  
    name VARCHAR(100) NOT NULL,  
    edit_date TIMESTAMP NOT NULL  
);
```

```
CREATE OR REPLACE FUNCTION fn_players_name_changes_log()  
    RETURNS TRIGGER  
    LANGUAGE plpgsql  
AS  
$$  
BEGIN  
    -- compare the new value vs the old value  
    -- OLD represents the row BEFORE the update  
    -- NEW represents the row that will be updated  
    IF NEW.name <> OLD.name THEN  
        INSERT INTO players_audit (player_id, name, edit_date)  
        VALUES (OLD.player_id, OLD.Name, NOW());  
    END IF;  
    RETURN NEW;  
END;  
$$
```

The trigger function is created. It detects if the old name is not equal to the new name and then inserts a new row into the player_audits table if true.

32.9 Bind a Function to a Table with a Trigger

```
CREATE TRIGGER trg_players_name_changes
BEFORE UPDATE
ON players
FOR EACH ROW
EXECUTE PROCEDURE fn_players_name_changes_log();
```

I bind the trigger function to the table.

```
INSERT INTO players (name) VALUES
('adam'),('linda');
```

I insert rows into the table but will it fire the trigger?
No because the trigger function is configured to only fire for UPDATE.

```
SELECT * FROM players;
```

	player_id [PK] integer	name character varying (100)
1	1	adam
2	2	linda

```
UPDATE players
SET name = 'Linda2'
WHERE player_id = 2;
```

If I update a name field in the players table it should be modified BUT also fire the trigger and add a record into the players_audit table.

```
SELECT * FROM players;
```

	player_id [PK] integer	name character varying (100)
1	1	adam
2	2	Linda2

```
SELECT * FROM players_audit;
```

	player_audit_id [PK] integer	player_id integer	name character varying (100)	edit_date timestamp without time zone
1	1	2	linda	2025-02-18 22:41:18.62914

```
UPDATE players
SET name = 'Linda3'
WHERE player_id = 2;
```

If I make another change the trigger is firing again and the trigger function inserts another row into the audits table.

```
SELECT * FROM players;
```

	player_id [PK] integer	name character varying (100)
1	1	adam
2	2	Linda3

```
SELECT * FROM players_audit;
```

	player_audit_id [PK] integer	player_id integer	name character varying (100)	edit_date timestamp without time zone
1	1	2	linda	2025-02-18 22:41:18.62914
2	2	2	Linda2	2025-02-18 22:45:10.082281

```
UPDATE players
SET name = 'Linda3'
WHERE player_id = 2;
```

If I run an update query but the NEW.name is the same as the OLD.name then no rows are added in the audit table because of the if clause in the trigger function.

```
SELECT * FROM players;
```

	player_id [PK] integer	name character varying (100)
1	1	adam
2	2	Linda3

```
SELECT * FROM players_audit;
```

	player_audit_id [PK] integer	player_id integer	name character varying (100)	edit_date timestamp without time zone
1	1	2	linda	2025-02-18 22:41:18.62914
2	2	2	Linda2	2025-02-18 22:45:10.082281

32.10 Modify Data at INSERT event

Triggers can modify data BEFORE or AFTER the actual modifications have happened. This is a good way to verify data and to error out data mistakes and more. To demonstrate this we will use a trigger to check inserted data then modify it if needed as per logic.

```
CREATE TABLE t_temperature_log (  
  id_temperature_log SERIAL PRIMARY KEY,  
  add_date TIMESTAMPTZ,  
  temperature NUMERIC  
);
```

A simple table that logs temperature and timestamp.

```
-- The INSERT and UPDATE always provide a NEW variable  
-- UPDATE and DELETE will offer a variable called OLD  
These variables contain the row that you are about to modify.
```

```
CREATE OR REPLACE FUNCTION fn_temperature_value_check_at_insert()  
  RETURNS TRIGGER  
  LANGUAGE plpgsql  
AS  
$$  
BEGIN  
  IF NEW.temperature < -30 THEN  
    NEW.temperature = 0;  
  END IF;  
  RETURN NEW;  
END;  
$$
```

A trigger function to check temperature and modify value if it is less than -30.

```
CREATE TRIGGER trg_temperature_log_insert_check
  BEFORE INSERT
  ON t_temperature_log
  FOR EACH ROW
  EXECUTE PROCEDURE fn_temperature_value_check_at_insert();
```

I bind the trigger function to the table.

```
INSERT INTO t_temperature_log (add_date, temperature)
VALUES ('2020-01-01', 10), ('2020-10-01', 20);
```

I insert some values that are above the threshold to be affected by the trigger.

```
SELECT * FROM t_temperature_log;
```

	id_temperature_log [PK] integer	add_date timestamp without time zone	temperature numeric
1	1	2020-01-01 00:00:00	10
2	2	2020-10-01 00:00:00	20
3	3	2020-01-01 00:00:00	10
4	4	2020-10-01 00:00:00	20

```
INSERT INTO t_temperature_log (add_date, temperature)
VALUES ('2020-01-01', -100);
INSERT 0 1
Query returned successfully in 96 msec.
```

Note how if I insert a value of temperature that is designed to fire the trigger, by default it completes the insert query with no warnings or notifications that the trigger has fired.

```
SELECT * FROM t_temperature_log;
```

	id_temperature_log [PK] integer	add_date timestamp without time zone	temperature numeric
1	1	2020-01-01 00:00:00	10
2	2	2020-10-01 00:00:00	20
3	3	2020-01-01 00:00:00	10
4	4	2020-10-01 00:00:00	20
5	5	2020-01-01 00:00:00	0

However, in the table the result of the trigger is that the NEW.temperature has been changed to the 0 as per the trigger function.

32.11 View Trigger Variables

Triggers Know a lot about themselves. They can access a couple of variables that allow you to write advanced code and more. Lets add a new function which displays some key trigger variables

```
CREATE OR REPLACE FUNCTION fn_trigger_variables_display()
RETURNS TRIGGER
LANGUAGE plpgsql
AS
$$
BEGIN
    RAISE NOTICE 'TG_NAME: %', TG_NAME;
    RAISE NOTICE 'TG_RELNAME: %', TG_RELNAME;
    RAISE NOTICE 'TG_TABLE_SCHEMA: %', TG_TABLE_SCHEMA;
    RAISE NOTICE 'TG_WHEN: %', TG_WHEN;
    RAISE NOTICE 'TG_LEVEL: %', TG_LEVEL;
    RAISE NOTICE 'TG_OP: %', TG_OP;
    RAISE NOTICE 'TG_NARGS: %', TG_NARGS;
    RAISE NOTICE 'TG_ARGV: %', TG_ARGV;
    RETURN NEW;
END;
$$
```

<https://www.postgresql.org/docs/current/plpgsql-trigger.html#~:text=41.10.1.%C2%A0Triggers%20on%20Data%20Changes%20%23>

```
CREATE TRIGGER trg_trigger_variables_display
AFTER INSERT
ON t_temperature_log
FOR EACH ROW
EXECUTE PROCEDURE fn_trigger_variables_display();
```

I bind the trigger function to the table as an AFTER INSERT event.


```

INSERT INTO t_temperature_log (add_date, temperature)
VALUES ('2020-01-01', -10);
NOTICE: TG_NAME: trg_trigger_variables_display
NOTICE: TG_RELNAME: t_temperature_log
NOTICE: TG_TABLE_SCHEMA: public
NOTICE: TG_WHEN: AFTER
NOTICE: TG_LEVEL: ROW
NOTICE: TG_OP: INSERT
NOTICE: TG_NARGS: 0
NOTICE: TG_ARGV: trg_trigger_variables_display
INSERT 0 1

Query returned successfully in 37 msec.

```

When I perform an insert operation this function is triggering and returning data about the operation.

This master function is useful for debugging.

32.12 Disallow DELETE on a Table

What if our business requirement is that the data can only be added and modified in some tables, but not deleted? One way to do this would be to just revoke the DELETE rights on these tables from all the users (remember to also revoke DELETE from PUBLIC), but this can also be achieved using triggers!!!

```

CREATE TABLE test_delete (
    id INT
);

```

```

INSERT INTO test_delete (id) VALUES (1),(2);

```

```

SELECT * FROM test_delete;

```

	id 
1	1
2	2

```

CREATE OR REPLACE FUNCTION fn_generic_cancel_op()
    RETURNS TRIGGER
    LANGUAGE plpgsql
AS
$$
BEGIN
    IF TG_WHEN = 'AFTER' THEN
        RAISE EXCEPTION 'YOU ARE NOT ALLOWED TO % ROWS IN % %',
            TG_OP, TG_TABLE_SCHEMA, TG_TABLE_NAME;
    END IF;
    RAISE NOTICE '% ON ROWS IN % % HAS NOT BEEN PERFORMED',
        TG_OP, TG_TABLE_SCHEMA, TG_TABLE_NAME;
    RETURN NULL;
END;
$$

```

Lets create a generic 'cancel' function.

This generic cancel function prevents The action passed (INSERT, DELETE, UPDATE, TRUNCATE).

```

CREATE TRIGGER trg_test_delete_disallow_delete
    AFTER DELETE
    ON test_delete
    FOR EACH ROW
    EXECUTE PROCEDURE fn_generic_cancel_op();

```

I now create a trigger for delete that is fired AFTER and bind it to the function.

```

DELETE FROM test_delete WHERE id = 1;
ERROR: YOU ARE NOT ALLOWED TO DELETE ROWS IN public test_delete
CONTEXT: PL/pgSQL function fn_generic_cancel_op() line 4 at RAISE
SQL state: P0001

```

When I try to delete rows I get the error message defined in the IF block of the generic cancel function.

```
SELECT * FROM test_delete;
```

	id integer 
1	1
2	2

Checking the tale I see that the row has not been deleted because the trigger has fired the generic cancel function AFTER.

```
CREATE TRIGGER trg__test_delete_skip_delete
BEFORE DELETE
ON test_delete
FOR EACH ROW
EXECUTE PROCEDURE fn_generic_cancel_op();
```

I now create a trigger for delete action that is fired BEFORE and bind it to the function.

```
DELETE FROM test_delete WHERE id = 1;
NOTICE: DELETE ON ROWS IN public test_delete HAS NOT BEEN PERFORMED
DELETE 0
```

Query returned successfully in 54 msec.

Because, in the generic cancel function, the IF block is set to fire on an AFTER action, and this trigger is set to BEFORE, the error message I get back is the one defined outside of the IF block in the trigger function.

```
SELECT * FROM test_delete;
```

	id integer 
1	1
2	2

Checking the table I see that the row has not been deleted because the trigger has fired the generic cancel function BEFORE.

The same trigger function can also be used to enforce a no-update policy or even a disallow inserts to a table that needs to have immutable content.

32.13 Disallow TRUNCATE on a Table

PostgreSQL TRUNCATE quickly removes all rows from a set of tables . It has the same effect as an unqualified DELETE on each table, but since it does not actually scan the tables, it is faster. Furthermore, it reclaims disk space immediately, rather than require a subsequent vacuum operation. This is most useful on large tables.

Lab scenario: In the previous example the user can still delete the record using a TRUNCATE! – TRUNCATE test_delete; so we will work on disallowing TRUNCATE actions.

Solution: We cannot simply skip TRUNCATE by returning NULL as opposed to the low-level BEFORE triggers. However, we can still make it impossible by raising an error if TRUNCATE is attempted.

```
CREATE TRIGGER trg_test_delete_disallow_truncate
  AFTER TRUNCATE
  ON test_delete
  FOR EACH ROW
  EXECUTE PROCEDURE fn_generic_cancel_op();
ERROR: TRUNCATE FOR EACH ROW triggers are not supported
SQL state: 0A000
```

Note the error. PostgreSQL will not allow me to place a AFTER TRUNCATE trigger to a row level action.

```
CREATE TRIGGER trg_test_delete_disallow_truncate
  AFTER TRUNCATE
  ON test_delete
  FOR EACH STATEMENT
  EXECUTE PROCEDURE fn_generic_cancel_op();
CREATE TRIGGER
Query returned successfully in 50 msec.
```

TRUNCATE ACTIONS are performed at the statement level.

```
TRUNCATE test_delete;  
ERROR: YOU ARE NOT ALLOWED TO TRUNCATE ROWS IN public test_delete  
CONTEXT: PL/pgSQL function fn_generic_cancel_op() line 4 at RAISE  
SQL state: P0001
```

Now the trigger is preventing the truncate action AFTER.

32.14 Creating an Audit Trigger

1. One of the most common uses of triggers is to log data changes to tables in a consistent and transparent manner.
2. When creating an audit trigger, we first must decide what we want to log.
3. A logical set of things that can be logged are:
 - Who changed the data,
 - When the data was changed,
 - Which operation changed the data

```
CREATE TABLE audit (  
    id INT  
);
```

```
CREATE TABLE audit_log(  
    username TEXT,  
    add_time TIMESTAMP,  
    table_name TEXT,  
    operation TEXT,  
    row_before JSON,  
    row_after JSON  
);
```

```

CREATE OR REPLACE FUNCTION fn_audit_trigger()
    RETURNS TRIGGER
    LANGUAGE plpgsql
AS
$$
DECLARE
    old_row json = NULL;
    new_row json = NULL;
BEGIN
    IF TG_OP IN ('UPDATE','DELETE') THEN
        -- Build old row
        old_row = row_to_json(OLD);
    END IF;
    IF TG_OP IN ('INSERT','UPDATE') THEN
        -- Build new row
        new_row = row_to_json(NEW);
    END IF;
    -- Insert data into audit_log
    INSERT INTO audit_log
    (
        username,
        add_time,
        table_name,
        operation,
        row_before,
        row_after
    )
    VALUES
    (
        SESSION_USER,
        CURRENT_TIMESTAMP AT TIME ZONE 'UTC',
        TG_TABLE_SCHEMA || '.' || TG_TABLE_NAME,
        TG_OP,
        old_row,
        new_row
    );
    RETURN NEW;
END;
$$

```

We will populate the audit_log table with the following:

Username	SESSION_USER
Add_time	Event time converted to UTC (Universal time coordinated) with daylight savings etc. CURRENT_TIMESTAMP AT TIMEZONE UTC
table_name	TG_TABLE_SCHEMA '.' TG_TABLE_NAME
operation	TG_OP (i.e. INSERT, UPDATE, DELETE)
row_before	Finally the row_before & row_after images of rows are stored as rows converted to JSON, which is available as it's own data type row_to_json.
row_after	

Note that the NEW and OLD are not NULL for the DELETE and INSERT triggers correspondingly.

```
CREATE TRIGGER trg_audit_trigger
AFTER INSERT OR UPDATE OR DELETE
ON audit
FOR EACH ROW
EXECUTE PROCEDURE fn_audit_trigger();
```

Bind the trigger to the trigger function and set it to fire on an INSERT, DELETE or UPDATE action

```
INSERT INTO audit (id) VALUES (1);
INSERT INTO audit (id) VALUES (2);
```

Insert some test values

The audit table contains the logs of who performed what action and when.

```
SELECT * FROM audit;
```

```
SELECT * FROM audit_log;
```

	id integer
1	1
2	2

Values are in the master table

	username text	add_time timestamp without time zone	table_name text	operation text	row_before json	row_after json
1	postgres	2025-02-19 18:50:16.205862	public.audit	INSERT	[null]	{"id":1}
2	postgres	2025-02-19 18:50:16.205862	public.audit	INSERT	[null]	{"id":2}

```
UPDATE audit
SET id = '100'
WHERE id = 1;
```

```
SELECT * FROM audit_log;
```

	username text	add_time timestamp without time zone	table_name text	operation text	row_before json	row_after json
1	postgres	2025-02-19 18:50:16.205862	public.audit	INSERT	[null]	{"id":1}
2	postgres	2025-02-19 18:50:16.205862	public.audit	INSERT	[null]	{"id":2}
3	postgres	2025-02-19 18:57:34.519049	public.audit	UPDATE	{"id":1}	{"id":100}

```
DELETE FROM audit
WHERE id = 2;
```

```
SELECT * FROM audit_log;
```

	username text	add_time timestamp without time zone	table_name text	operation text	row_before json	row_after json
1	postgres	2025-02-19 18:50:16.205862	public.audit	INSERT	[null]	{"id":1}
2	postgres	2025-02-19 18:50:16.205862	public.audit	INSERT	[null]	{"id":2}
3	postgres	2025-02-19 18:57:34.519049	public.audit	UPDATE	{"id":1}	{"id":100}
4	postgres	2025-02-19 19:00:37.201541	public.audit	DELETE	{"id":2}	[null]

NOTE: Triggers are called in ALPHABETICAL ORDER, and the latter triggers can change what gets inserted, so caution needs to be applied when naming triggers, particularly when auditing.

32.15 Creating Conditional Triggers

1. We can create a conditional trigger by using a generic WHEN clause
2. With a WHEN clause, you can write some conditions except a sub query (because sub query is tested before trigger function is called)
3. The WHEN expression should result into Boolean value. If the value is FALSE or NULL (which is automatically converted to FALSE), the trigger function is not called.

Lets say we want to enforce "No INSERT/UPDATE/DELETE on a Friday afternoon"!

We will use:

- 1. when condition
- 2. FOR EACH STATEMENT
- 3. Pass a 'parameter' to EXECUTE PROCEDURE(parameter)

```
CREATE TABLE mytask (  
    task_id SERIAL PRIMARY KEY,  
    task TEXT  
);
```



```
CREATE OR REPLACE FUNCTION fn_cancel_with_message()
    RETURNS TRIGGER
    LANGUAGE plpgsql
AS
$$
BEGIN
    RAISE EXCEPTION TG_ARGV[0];
    RETURN NULL;
END;
$$
ERROR: syntax error at or near "TG_ARGV"
LINE 7: RAISE EXCEPTION TG_ARGV[0];
                                ^

SQL state: 42601
Character: 126
```

Corected trigger function where
TG_ARGV is supplied with text '%'.
^

I have created a generic function that
returns a message based on TG_ARGV
but It fails because TG_ARGV needs to
be supplied with text.

```
CREATE OR REPLACE FUNCTION fn_cancel_with_message()
    RETURNS TRIGGER
    LANGUAGE plpgsql
AS
$$
BEGIN
    RAISE EXCEPTION '%', TG_ARGV[0];
    RETURN NULL;
END;
$$
CREATE FUNCTION
Query returned successfully in 42 msec.
```

```
CREATE TRIGGER trg_no_update_on_friday_afternoon
    BEFORE INSERT OR UPDATE OR DELETE OR TRUNCATE
    ON mytask
    FOR EACH STATEMENT
    WHEN
    (
        EXTRACT('DOW' FROM CURRENT_TIMESTAMP) = 5
        AND CURRENT_TIME > '12:00'
    )
    EXECUTE PROCEDURE fn_cancel_with_message('No
updates on Friday aftrnoon');
```

I bind the trigger function to the table
at statement level using an additional
when clause.

Note I am also passing a string as a
parameter to the trigger function which
will be intercepted as TG_ARGV[0].

```
SELECT  
EXTRACT('DOW' FROM CURRENT_TIMESTAMP),  
CURRENT_TIME;
```

	extract numeric	current_time time with time zone
1	3	20:55:59.987701+01:00

The current day of the week is Wednesday and the time is 20:55 so my previously created trigger would not fire. To test its functionality, I can modify it to fire for Wednesday.

```
CREATE OR REPLACE TRIGGER trg_no_update_on_friday_afternoon  
  BEFORE INSERT OR UPDATE OR DELETE OR TRUNCATE  
  ON mytask  
  FOR EACH STATEMENT  
  WHEN  
  (  
    EXTRACT('DOW' FROM CURRENT_TIMESTAMP) = 3  
    AND CURRENT_TIME > '12:00'  
  )  
  EXECUTE PROCEDURE fn_cancel_with_message('No updates on Wednesday aftrnoon');
```

```
INSERT INTO mytask (task) VALUES ('task1');  
ERROR: No updates on Wednesday aftrnoon  
CONTEXT: PL/pgSQL function fn_cancel_with_message() line 3 at RAISE  
SQL state: P0001
```

The trigger works.
It is meeting the
condition in the
WHEN clause.

32.16 Disallow data change on Primary Key

```
CREATE TABLE pk_table (  
    id SERIAL PRIMARY KEY,  
    t TEXT  
);
```

```
INSERT INTO pk_table (t)  
VALUES ('t1'),('t2');
```

```
SELECT * FROM pk_table;
```

	id [PK] integer	t text
1	1	t1
2	2	t2

```
CREATE OR REPLACE FUNCTION fn_generic_cancel_op()  
    RETURNS TRIGGER  
    LANGUAGE plpgsql  
AS  
$$  
BEGIN  
    IF TG_WHEN = 'AFTER' THEN  
        RAISE EXCEPTION 'YOU ARE NOT ALLOWED TO % ROWS IN % %',  
            TG_OP, TG_TABLE_SCHEMA, TG_TABLE_NAME;  
    END IF;  
    RAISE NOTICE '% ON ROWS IN % % HAS NOT BEEN PERFORMED',  
        TG_OP, TG_TABLE_SCHEMA, TG_TABLE_NAME;  
    RETURN NULL;  
END;  
$$
```

I am going to reuse the
previously created
generic_cancel_op function

```
CREATE TRIGGER disallow_pk_change  
    AFTER UPDATE OF id  
    ON pk_table  
    FOR EACH ROW  
    EXECUTE PROCEDURE fn_generic_cancel_op();
```

I am going to use an AFTER UPDATE trigger
and bind it to the id column (using OF id) of
the pk_table.

```
UPDATE pk_table SET id = 100 WHERE id = 1;  
ERROR: YOU ARE NOT ALLOWED TO UPDATE ROWS IN public pk_table  
CONTEXT: PL/pgSQL function fn_generic_cancel_op() line 4 at RAISE  
SQL state: P0001
```

The trigger stops
the update.

32.17 Use Triggers Very Cautiously

Yes, triggers are powerful tools, as they can be used for

- Auditing
- Logging
- Enforcing complex constraints
- Replications and more

However, if they are not used cautiously, they will be very hard to debug problems!!!

Some good practices to follow here are:

1. Do not change data in Primary key, Foreign Key or unique key columns
2. Do not update records in the tables that you normally read during the transaction
3. Do not read data from a table that is updating during the same transaction
4. Do not aggregate/summarise over the table that you are updating

32.18 What is an Event Trigger

1. Event triggers are data specific and not bound to or attached to a table.
2. Unlike regular triggers, they capture system level DDL events.
3. Event triggers can be BEFORE or AFTER triggers.
4. Trigger functions can be written in any language except SQL
5. Event triggers are disabled in the singular user mode and can only be created by a superuser.

32.19 Event Triggers Usage Scenarios

1. Audit trail – Users, schema changes etc
2. Preventing changes to a table.
3. Providing limited DDL (Data definition Language) capabilities to developers.
4. Enable/Disable certain DDL commands based on business logic.
5. Performance analysis of DDL start (ddl_command_start) and end (DDL_command_end) process.
6. Replicating DDL changes/updates to remote locations
7. Cascading a DDL.

32.20 Creating Event Triggers

CREATE EVENT TRIGGER trg_name

Requirements

1. Before using an event trigger, we must have a function that the trigger will execute
2. The function must return a special type called EVENT_TRIGGER
3. This function need not (and may not) return a value. The return type serves merely as a signal that the function is to be invoked as an event trigger.

In case of multiple triggers

They are executed in alphabetical order of their names

Can we create conditional event triggers?

Yes, you can create conditional event triggers by using the WHEN clause

```
CREATE EVENT TRIGGER trg_name  
WHEN
```

```
    <condition>
```

Event triggers (like other functions) cannot be executed in an aborted transaction

i.e. if a DDL command fails with an error, any associated ddl_command_end triggers will not be executed.

Conversely, if a ddl_command_start trigger fails with an error, no further event triggers will fire, and no attempt will be made to execute the command itself.

Similarly, if a ddl_command_trigger fails with an error, the effects of the DDL statement will be rolled back, just as they would be in any other case where the containing transaction aborts.

32.21 Event Trigger Events & Variables

Event triggers events

ddl_commands_start	This event occurs just BEFORE a CREATE, ALTER or DROP DDL command is executed.
ddl_command_end	This event occurs just AFTER a CREATE, ALTER or DROP command has FINISHED executing.
Table_rewrite	This event occurs just before a table is rewritten by some actions of the commands ALTER TABLE and ALTER TYPE.
Sql_drop	This event occurs just before the ddl_command_end event for the commands that drop database objects.

Event trigger variables

TG_TAG	This variable contains the 'TAG' of the command for which the trigger is executed This variable does not contain the full command string, but just a tag such as CREATE TABLE, DROP TABLE, ALTER TABLE and so on.
TG_EVENT	This variable contains the event name, which can be ddl_command_start, and sql_drop

32.22 Creating and Audit Event Trigger

```
CREATE TABLE audit_ddl (  
    audit_ddl_id SERIAL PRIMARY KEY,  
    username TEXT,  
    ddl_event TEXT,  
    ddl_command TEXT,  
    ddl_add_time TIMESTAMPTZ  
);
```

1. Lets create a table that will capture DDL events and log these events (such as table creation, modification and drop.)

2. Create the event trigger function. Note that it RETURNS EVENT TRIGGER.

The SECURITY DEFINER line is the ensure that this function is created from a superuser level.

The event trigger function does not return a value.

```
CREATE OR REPLACE FUNCTION fn_event_audit_ddl()  
    RETURNS EVENT_TRIGGER  
    LANGUAGE plpgsql  
    SECURITY DEFINER --  
  
AS  
$$  
BEGIN  
    -- INSERT into the audit DDL table  
    INSERT INTO audit_ddl  
    (  
        username,  
        ddl_event,  
        ddl_command,  
        ddl_add_time  
    )  
    VALUES  
    (  
        SESSION_USER,  
        TG_EVENT,  
        TG_TAG,  
        NOW()  
    );  
    -- RAISE NOTICE  
    RAISE NOTICE 'DDL activity is logged';  
    -- No return value is needed  
  
END;  
$$
```



```
CREATE EVENT TRIGGER trg_event_audit_ddl
ON ddl_command_start
EXECUTE PROCEDURE fn_event_audit_ddl();
```

```
CREATE EVENT TRIGGER trg_event_audit_ddl
ON ddl_command_start
WHEN
    TAG IN('CREATE TABLE')
EXECUTE PROCEDURE fn_event_audit_ddl();
```

```
SELECT * FROM audit_ddl;
```

The audit_ddl log table is currently empty.

	audit_ddl_id [PK] integer	username text	ddl_event text	ddl_command text	ddl_add_time timestamp with time zone

```
CREATE TABLE audit_ddl_test (
    i INT
);
```

**NOTICE: DDL activity is logged
CREATE TABLE**

Query returned successfully in 45 msec.

```
SELECT * FROM audit_ddl;
```

	audit_ddl_id [PK] integer	username text	ddl_event text	ddl_command text	ddl_add_time timestamp with time zone
1	1	postgres	ddl_command_start	CREATE TABLE	2025-02-19 23:15:06.122986+01

```
DROP EVENT TRIGGER trg_event_audit_ddl;
```

3. Bind the trigger function. Note that the trigger function is not bound to a table but is bound to an event trigger events. i.e. ddl_command_start.

This event trigger can be further enhanced to include a conditional WHEN to target a particular event tag.

4. When I create a table I get a custom message, that is defined in the event trigger function.

The audit_ddl log table now contains an entry logging this create table event.

If I want to modify an event trigger, then I have to drop it and recreate it.

32.23 Prevent Schema Changes

Lab Scenario: implement a policy so that no tables are allowed to be created between 9am and 4pm.

```
CREATE OR REPLACE FUNCTION fn_event_abort_create_table_function()  
    RETURNS EVENT_TRIGGER  
    LANGUAGE plpgsql  
    SECURITY DEFINER  
AS  
$$  
DECLARE  
    current_hour int = EXTRACT('hour' FROM NOW());  
BEGIN  
    IF current_hour BETWEEN 9 AND 16 THEN  
        RAISE EXCEPTION 'Tables are not allowed to be created during 9am to 4pm';  
    END IF;  
END;  
$$
```

```
CREATE EVENT TRIGGER trg_evt_abort_create_table_function  
ON ddl_command_start  
WHEN  
    TAG IN ('CREATE TABLE')  
EXECUTE PROCEDURE fn_event_abort_create_table_function();
```

```
CREATE TABLE ddl_t3 (i INT);  
ERROR: 'Tables are not allowed to be created during 9am to 4pm'  
CONTEXT: PL/pgSQL function fn_event_abort_create_table_function() line 10 RAISE  
SQL STATE: P0001
```

32.24 Dropping a Trigger

```
DROP EVENT TRIGGER trg_evnt_abort_create_table_function;
```

Housekeeping. Dropping this trigger so that tables can be created between 9am and 4pm in the future.

33. Using Cursors

[33.1 Understanding Row by Row Operations](#)

[33.2 Cursors and Procedural Language](#)

[33.3 Steps to Create a Cursor](#)

[33.4 Creating a Cursor](#)

[33.5 Opening a Cursor](#)

[33.6 Using a Cursor](#)

[33.7 Updating Data](#)

[33.8 Closing a Cursor](#)

[33.9 Creating a PL/pgSQL Cursor](#)

[33.10 Using a Parametric Cursor via Function](#)

33.1 Understanding Row by Row Operations

1. SQL retrieval operations work sets of rows as 'results sets' or 'dataset'.
2. The rows returns are all the rows that return s match a SQL statement, Either zero or more of them
3. E.g. when you use a simple SELECT statement, there is no way to get the first row, the next row, or the previous five rows etc.. And this is how relational DBMS's work.
4. Traditional procedural languages like Python, php, etc can operate row by row.
5. Sometimes you need to step through rows FORWARD or BACKWARD and ONE or MORE AT A TIME. This is what 'cursor' is used for.
6. A cursor enables SQL to retrieve (or update, or delete) a single row at a time.
7. A cursor is like a pointer that points or locates a specific table row.
8. A cursor is a database query store on the DBMS server, not a select statement. But it is the result set retrieved by that statement.
9. Once the cursor is active or stored, you can;
 - SELECT
 - UPDATE
 - DELETEA row at which the cursor is pointing.

33.2 Cursors and Procedural Language

1. Cursors are very valuable if you want to retrieve SELECTED ROWS from a table.
2. Once retrieved, you can check the contents of the results sets, and perform different operations on those contents.
3. However, please note SQL cannot perform this sequence of operations by itself. SQL can retrieve the rows, but ideally the operations should be done by procedural languages based on content.
4. Procedural and non-procedural languages.

In PROCEDURAL languages, the program code is written as a sequence of instructions. User has to specify "what to do" and also "how to do" (step by step procedure). These instructions are executed in sequential order. These instructions are written to solve specific problems.

Examples of procedural languages: FORTRAN, COBOL, ALGOL, BASIC, C, pascal etc

In NON-PROCEDURAL languages, the user has to specify only "what to do" and not "how to do it". It is also known as applicative or functional language. It involves the development of the functions from other functions to construct more complex functions.

Examples of non-Procedural languages: SQL, PROLOG, LISP

5. Cursors can retrieve and then feed the contents/results sets to a procedural language for further processing.

33.3 Steps to Create a Cursor

Before using a cursor the following steps are required:

1. DECLARE

A cursor must be DECLARE before it is to be used. This does not retrieve any data, it just defines the select statement to be used and any other cursor options.

2. OPEN

Once it is declare, it must be OPENed for use. This process retrieves the data using the define SELECT statement.

3. FETCH

Within the cursor, populate with data, individual rows can be fetched (retrieved as per needed).

4. CLOSE

When you do not need the cursor, it must be closed. This operation then deallocates memory etc back to the DMBS.

5. Once a cursor is declared it may be opened, fetched or closed as often as needed.

6. So the Steps are DECLARE → OPEN → FETCH → CLOSE

33.4 Creating a Cursor

1. Declaring a cursor using refcursor data type. PostgreSQL provides you with a special type called REFCURSOR to declare a cursor variable.

```
DECLARE cursor_name refcursor;
```

Cursor_name uniquely identifies a cursor in the current module or compilation unit.

DECLARE cursor_name refcursor; - This will create a cursor variable. This will not work because all the cursors need to be created in a transaction block.

2. Create a cursor that binds to a query expression

```
cursor_name [cursor_scrollability] CURSOR [(name datatype, name datatype ...)]  
FOR
```

Query-expression

cursor-scrollability SCROLL or NO SCROLL (default) NO SCROLL means that the cursor cannot scroll backwards.

Query_expression You can use any legal SELECT statement as a query expression. The results sets rows are considered as scope of the cursor.

```
DECLARE  
  cur_all_movies CURSOR  
  FOR  
  SELECT movie_name, movie_length FROM movies;
```


3. Create a cursor with query parameters

DECLARE

cur_all_movies_by_year CURSOR (custom_year INTEGER)

FOR

SELECT movie_name, movie_length

FROM movies

WHERE EXTRACT('year' FROM release_date) = custom_year;

33.5 Opening a Cursor

1. Unbound cursor is not bound to any query expressions.

OPEN unbound_cursor_variable [[No] SCROLL] FOR Query;

SELECT * FROM directors;

OPEN cur_directors_us

FOR

SEELCT

first_name,

Last_name,

date_of_birth

FROM directors

WHERE nationality = 'US';

2. Opening an unbound cursor with dynamic query

```
OPEN unbound_cursor_variable [ [ No ] SCROLL ]  
FOR EXECUTE  
    query-expression [using expression [, ...] ];
```

```
My_query = 'SELECT DISTINCT (nationality) FROM directors ORDER BY $1';
```

```
OPEN cur_directors_nationality  
FOR EXECUTE  
    my_query USING sort_fields;
```

3. Opening a bound cursor. As they are bound to a query when we declare it, so when we open it, we just need to pass the arguments to the query if necessary.

```
OPEN cursor_variable [ (name:=value, name:=value)];
```

```
OPEN cursor_all_movies;
```

```
OPEN cur_all_movies_by_year (custom_year:=2010);
```

33.6 Using a Cursor

1. Following operations can be done once a cursor is open

FETCH, MOVE, UPDATE or DELETE statement.

2. **FETCH statement**

```
FETCH [ direction { FROM | IN } ] cursor_variable INTO target_variable;
```

```
FETCH cur_all_movies INTO row_movie;
```

3. **Direction.** By default, a cursor gets the next row if you don't specify the direction explicitly.

```
NEXT    LAST    PRIOR    FIRST    ABSOLUTE count    RELATIVE count    FORWARD    BACKWARD
```

```
FETCH LAST
```

```
FROM row_movie
```

```
INTO movie_title,movie_release_year;
```

If you enable **SCROLL** at the declaration of the cursor, you can only use **FORWARD** or **BACKWARD**

4. **Moving the cursor** If you want to move the cursor only without retrieving any row, you will use the move statement.

```
MOVE [ direction { FROM | IN } ] cursor_variable;
```

```
MOVE cur_all_movies; will move the cursor to the next one
```

```
MOVE LAST FROM cur_all_movies; Will move the cursor to the last
```

```
MOVE relative -1 FROM cur_all_movies; Will move the cursor back 1
```

```
MOVE FORWARD FROM cur_all_movies; Will move to the cursor to the next one
```

33.7 Updating Data

Once a cursor is positioned, we can delete or update row identified by the cursor using the following statement.

```
DELETE WHERE CURRENT OF  
UPDATE WHERE CURRENT OF
```

```
UPDATE movies  
SET YEAR (release_date) = custom_year  
WHERE  
    current of cur_all_movies;
```

33.8 Closing a Cursor

Close statement releases resources or frees up cursor variable to allow it to be opened again using open statement.

```
CLOSE cursor_variable;
```

```
CLOSE cur_all_movies;
```

```
OPEN cur_all_movies;
```

33.9 Creating a PL/pgSQL Cursor

```
DO
$$
DECLARE
    output_text TEXT DEFAULT '';
    rec_movie record;
    cur_all_movies CURSOR
    FOR
        SELECT * FROM movies;
BEGIN
    OPEN cur_all_movies;

    LOOP
        FETCH cur_all_movies INTO rec_movie;
        EXIT WHEN NOT FOUND;
        output_text := output_text || '|' || rec_movie.movie_name;
    END LOOP;
    RAISE NOTICE 'ALL MOVIES NAMES %', output_text;
END;
$$
```

```
NOTICE: ALL MOVIES NAMES |A Clockwork Orange|Apocalypse Now|Battle Royale |Blade
Runner |Chungking Express|City of God|City of Men|Cold Fish|Crouching Tiger Hidden
Dragon|Eyes Wide Shut|Forrest Gump|Gladiator|Gone with the Wind|Goodfellas|Grand
Budapest Hotel|House of Flying Daggers|In the Mood for Love|Jaws|Leon|Let the Right
One In|Life of Brian|Life of Pi|Mary Poppins|Never Let Me Go|Oldboy|Pans
Labyrinth|Ponyo|Pulp Fiction|Raging Bull|Rushmore|Spider-Man|Spider-Man 2|Spider-Man
3|Spirited Away|Star Wars: A New Hope|Star Wars: Empire Strikes Back|Star Wars:
Return of the Jedi|Submarine|Taxi Driver|The Darjeeling Limited|The Fifth Element|The
Lives of Others|The Shining|The Sound of Music|The Wizard of Oz|There Will Be
Blood|Three Billboards Outside Ebbing, Missouri |Titanic|Top Gun|Toy Story|V for
Vendetta|Watchmen|Way of the Dragon
DO
Query returned successfully in 36 msec.
```

33.10 Using a Parametric Cursor via Function

Lets create a function where we will use the cursor to loop through all the movies and concatenate movie title and release date of movies that contain the word 'star' in the title.

```
CREATE OR REPLACE FUNCTION fn_get_movie_names_by_year(custom_year integer)
RETURNS TEXT
LANGUAGE plpgsql
AS
$$
DECLARE
    movie_names TEXT DEFAULT '';
    rec_movie record;
    cur_all_movies_by_year CURSOR (custom_year integer)
    FOR
        SELECT
            movie_name,
            EXTRACT('YEAR' FROM release_date) AS release_year
        FROM movies
        WHERE EXTRACT('YEAR' FROM release_date) = custom_year;
BEGIN
    OPEN cur_all_movies_by_year(custom_year);
    LOOP
        FETCH cur_all_movies_by_year INTO rec_movie;
        EXIT WHEN NOT FOUND;
        IF rec_movie.movie_name LIKE '%star%' THEN
            movie_names := movie_names || ',' || rec_movie.movie_name || ':' ||
rec_movie.release_year;
        END IF;
    END LOOP;
    CLOSE cur_all_movies_by_year;
    RETURN movie_names;
END;
$$
```

```
SELECT fn_get_movie_names_by_year(1977);
```

34. Creating Crosstab Reports

[34.1 What is a Crosstab Report?](#)

[34.2 Installing the tablefunc Extension](#)

[34.3 Creating a Crosstab Report](#)

[34.4 Orders Matter in Crosstab Reports](#)

[34.5 Pivoting with Rainfalls Data](#)

[34.6 Pivoting Rows and Columns](#)

[34.7 Matrix Report via a Query](#)

[34.8 Aggregate over a Filter](#)

[34.9 Static to Dynamic Pivots](#)

[33.10 Dynamic Pivot query via JSON](#)

[33.10 Dynamic Pivot Table Columns](#)

[33.11 Interactive Client-side Pivot](#)

[33.12 Handling missing Values](#)

34.1 What is a Crosstab Report?

1. A typical relational database table contains multiple rows, quite often with duplicate values in some columns. Data in such a table is usually stored in random order. By running a query to select data from a database table, you can perform filtering, sorting, grouping, selection and other operations with data.
2. Nevertheless, the results of that kind of Query (data) will still be displayed downwards, which complicates the analysis. Pivot tables (inverted tables) extend the data across rather than downward. In this way, the query results are much easier to perceive, compare, analyse and filter.
3. A 'pivot' table or 'crosstab report' is an effective technique for calculating, compiling, and analysing data bound to simplify the search for patterns and trends. Pivot tables can help aggregate, sort, organise, recognise, group, sum or average data stored in a database to understand data relations and dependencies in the best possible way.
4. Pivot table/Crosstab report/Cross Tabulation provide a simple way to summarise and compare variables by displaying them in a table layout or in a matrix format.
5. In a matrix, Rows represent ONE variable, Columns represent another variable. Cell represents where a row and column intersect to hold a value.

34.2 Installing the tablefunc Extension

In order to use the crosstab function, an extension needs to be installed called 'tablefunc'.

```
SELECT * FROM pg_extension;
```

	oid [PK] oid	extname name	extowner oid	extnamespace oid	extrelocatable boolean	extversion text	extconfig oid[]	extcondition text[]
1	15000	plpgsql	10	11	false	1.0	[null]	[null]

```
CREATE EXTENSION IF NOT EXISTS tablefunc;
```

```
SELECT * FROM pg_extension;
```

	oid [PK] oid	extname name	extowner oid	extnamespace oid	extrelocatable boolean	extversion text	extconfig oid[]	extcondition text[]
1	15000	plpgsql	10	11	false	1.0	[null]	[null]
2	132033	tablefunc	10	2200	true	1.0	[null]	[null]

<https://www.postgresql.org/docs/current/tablefunc.html>

34.3 Creating a Crosstab Report

```
CREATE TABLE scores (  
    score_id SERIAL PRIMARY KEY,  
    name VARCHAR(100),  
    subject VARCHAR(100),  
    score NUMERIC(4,2),  
    score_date DATE  
);
```

```
SELECT * FROM scores;
```

```
INSERT INTO scores (name, subject, score, score_date)  
VALUES  
( 'Adam', 'Math', 10, '2000-01-01'),  
( 'Adam', 'English', 8, '2000-02-01'),  
( 'Adam', 'History', 8, '2000-03-01'),  
( 'Adam', 'Music', 8, '2000-04-01'),  
( 'Linda', 'Math', 12, '2000-01-01'),  
( 'Linda', 'English', 10, '2000-02-01'),  
( 'Linda', 'History', 8, '2000-03-01'),  
( 'Linda', 'Music', 6, '2000-04-01');
```

	score_id [PK] integer	name character varying (100)	subject character varying (100)	score numeric (4,2)	score_date date
1	1	Adam	Math	10.00	2000-01-01
2	2	Adam	English	8.00	2000-02-01
3	3	Adam	History	8.00	2000-03-01
4	4	Adam	Music	8.00	2000-04-01
5	5	Linda	Math	12.00	2000-01-01
6	6	Linda	English	10.00	2000-02-01
7	7	Linda	History	8.00	2000-03-01
8	8	Linda	Music	6.00	2000-04-01

Name	Math	English	History	Music
Adam	10	8	8	8
Linda	12	10	8	6

The objective is to convert the table from a normal row output as above into a pivot table /crosstab report like shown to the left.

1. We will use the cross tab function here, where the syntax is:

```
SELECT * FROM crosstab
(
    select_query_expression
) AS alias_name
(
    col1 type,
    col2 type,
    ...
)
```

2. The select must return 3 columns, (X, Y, V)

IDENTIFIER	Rows	Y
CATEGORIES	Columns	X
VALUE	Cell	V

3. The first column in the SELECT will be the IDENTIFIER of every row in the pivot table or final results.

4. The second column in the SELECT represents the CATEGORIES in the pivot table

5. The third column in the SELECT represents the VALUE to be assigned to each cell of the pivot table.

So our Identifier Categories Values (ICV) here needs to be:

Y: name,

X: subject,

V: score

```

SELECT * FROM crosstab
(
    SELECT
        name,
        subject,
        score
    FROM scores
) AS ct
(
    name varchar,
    English numeric,
    History numeric,
    Math numeric,
    Music numeric
);

```

	name character varying	english numeric	math numeric	history numeric	music numeric
1	Adam	10.00	8.00	8.00	8.00
2	Linda	12.00	10.00	8.00	6.00

Within single quotes, I select the rows from the source table where I want the data to come from. This is a crosstab query.

I then define my new columns for the cross tab report making sure the datatype of the column matches that datatype that is in the row of the source table. For example if name was TEXT and not VARCHAR it would not work.

Note how the readability of the crosstab report is much better compared to the raw data from the source table.

34.4 Orders Matter in Crosstab Reports

```

SELECT
name,
subject,
score
FROM scores

```

	name character varying (100)	subject character varying (100)	score numeric (4,2)
1	Adam	Math	10.00
2	Adam	English	8.00
3	Adam	History	8.00
4	Adam	Music	8.00
5	Linda	Math	12.00

However, all is not well. If I review the source table I see that Adam scored 10 in Math, yet in the above cross tab output I got a value of 8.

```
SELECT * FROM crosstab
(
    ,
    SELECT
        name,
        subject,
        score
    FROM scores
    ,
) AS ct
(
    name varchar,
    English numeric,
    History numeric,
    Math numeric,
    Music numeric
);
```

The reason for this is that I defined the crosstab report columns alphabetically, yet they do not appear in the source table alphabetically and so are mapped across incorrectly.

```
SELECT
name,
subject,
score
FROM scores
ORDER BY 1,2;
```

If I order the data in the source table first by name then by subject.

	name character varying (100)	subject character varying (100)	score numeric (4,2)
1	Adam	English	8.00
2	Adam	History	8.00
3	Adam	Math	10.00
4	Adam	Music	8.00
5	Linda	English	10.00
6	Linda	History	8.00
7	Linda	Math	12.00
8	Linda	Music	6.00

```
SELECT * FROM crosstab
(
    ,
    SELECT
        name,
        subject,
        score
    FROM scores
    ORDER BY 1,2
    ,
) AS ct
(
    name varchar,
    English numeric,
    History numeric,
    Math numeric,
    Music numeric
);
```

The pivot table now shows the correct data in the columns of each subject. **Order matters in crosstab reports.**

	name character varying 	english numeric 	history numeric 	math numeric 	music numeric 
1	Adam	8.00	8.00	10.00	8.00
2	Linda	10.00	8.00	12.00	6.00

34.5 Pivoting with Rainfalls Data

The rainfalldata.sql contains some dummy data to play with.

```
movies-# \i C:/Users/ellio/Documents/CODING-LESSONS/15-Advanced_PostgreSQL/resources/rainfalldata.sql
CREATE TABLE
COPY 648
movies-#
movies=# SELECT * FROM rainfalls;
 location | year | month | rainedays
-----+-----+-----+-----
 Germany  | 2017 |      1 |         9
 Germany  | 2017 |      2 |        10
 ...
 Qatar    | 2012 |     10 |         8
 Qatar    | 2012 |     11 |         5
 Qatar    | 2012 |     12 |        12
(648 rows)
```

```
1  --
2  -- Sample dummy data for rainfalls
3  --
4
5  CREATE TABLE rainfalls (
6      location text,
7      year integer,
8      month integer,
9      rainedays integer
10 );
11
12 COPY rainfalls (location, year, month, rainedays) FROM stdin;
Germany 2017      1      9
Germany 2017      2     10
Germany 2017      3      9
Germany 2017      4      2
```

Which I import from a terminal and verify that I have a table and data.

Lets build a pivot table to display a sum of all rain days per each location for each year. Therefore:

X	years	(INTEGER)
Y	location	(TEXT)
Z	sum(rainedays)	(INTEGER)

	location 	year 	month 	rainedays 
1	Germany	2017	1	9
2	Germany	2017	2	10

```
SELECT * FROM crosstab  
(
```

```
SELECT  
  location,  
  year,  
  sum(raindays)  
FROM rainfalls  
GROUP BY location, year  
ORDER BY location, year
```

```
) AS ct
```

```
(  
  location text,  
  "2012" bigint,  
  "2013" bigint,  
  "2014" bigint,  
  "2015" bigint,  
  "2016" bigint,  
  "2017" bigint  
);
```

	location text	2012 bigint	2013 bigint	2014 bigint	2015 bigint	2016 bigint	2017 bigint
1	Dubai	112	116	111	80	110	102
2	France	114	111	113	85	120	110
3	Germany	153	120	136	128	138	113
4	London	116	138	137	101	117	110
5	Malaysia	53	77	78	50	52	43
6	Qatar	86	116	111	83	102	89
7	Singapore	124	132	142	111	106	110
8	Sydney	98	120	110	93	122	115
9	Tokyo	47	63	68	53	54	43

Note that the return type of sum(raindays) is bigint, not int. If I was to define the output table with int values then it would throw an error.

34.6 Pivoting Rows and Columns

```
SELECT * FROM crosstab
(
    ,
    SELECT
        location, month, sum(raindays)::int
    FROM rainfalls
    GROUP BY location, month
    ORDER BY 1,2
    ,
) AS ct
(
    location text,
    "jan" int,
    "feb" int,
    "mar" int,
    "apr" int,
    "may" int,
    "jun" int,
    "jul" int,
    "aug" int,
    "sep" int,
    "oct" int,
    "nov" int,
    "dec" int
);
```

With the same rainfalls dataset I can put total rainfall days per month for all the years.

Note how I am typecasting the sum(raindays) to an integer so I can use integer datatype in the output table.

	location 	jan integer	feb integer	mar integer	apr integer	may integer	jun integer	jul integer	aug integer	sep integer	oct integer	nov integer	dec integer
1	Dubai	60	60	48	59	71	57	49	45	35	47	57	43
2	France	64	54	44	49	70	56	43	45	49	56	60	63
3	Germany	81	67	52	55	66	68	61	62	50	73	84	69
4	London	91	75	64	63	57	51	40	35	51	53	74	65
5	Malaysia	40	45	39	36	27	16	7	10	23	33	47	30
6	Qatar	63	58	60	57	64	42	38	40	33	28	62	42
7	Singapore	90	78	57	47	60	48	46	39	41	68	74	77
8	Sydney	70	52	38	42	69	62	58	44	48	51	67	57
9	Tokyo	33	39	35	37	24	13	10	20	24	29	41	23


```

SELECT * FROM crosstab
(
    ,
    SELECT
        month,location, sum(raindays)::int
    FROM rainfalls
    GROUP BY month, location
    ORDER BY 1,2
    ,
) AS ct
(
    month int,
    "Dubai" int,
    "France" int,
    "Germany" int,
    "London" int,
    "Malaysia" int,
    "Qatar" int,
    "Singapore" int,
    "Sydney" int,
    "Tokyo" int
);

```

This time I am pivoting the data to have countries (location) as column headers and months as rows.

	month integer	Dubai integer	France integer	Germany integer	London integer	Malaysia integer	Qatar integer	Singapore integer	Sydney integer	Tokyo integer
1	1	60	64	81	91	40	63	90	70	33
2	2	60	54	67	75	45	58	78	52	39
3	3	48	44	52	64	39	60	57	38	35
4	4	59	49	55	63	36	57	47	42	37
5	5	71	70	66	57	27	64	60	69	24
6	6	57	56	68	51	16	42	48	62	13
7	7	49	43	61	40	7	38	46	58	10
8	8	45	45	62	35	10	40	39	44	20
9	9	35	49	50	51	23	33	41	48	24
10	10	47	56	73	53	33	28	68	51	29
11	11	57	60	84	74	47	62	74	67	41
12	12	43	63	69	65	30	42	77	57	23

34.7 Matrix Report via a Query

```
SELECT
  y,
  aggregate_function (CASE WHEN x = 'value 1' THEN v END) "value 1",
  aggregate_function (CASE WHEN x = 'value 2' THEN v END) "value 2",

  repeat for each x transposed into a column

FROM table or subquery
GROUP BY y
[ORDER BY 1]
```

```
SELECT
name,
(CASE WHEN subject = 'English' THEN score END) as English,
(CASE WHEN subject = 'History' THEN score END) as History,
(CASE WHEN subject = 'Math' THEN score END) as Math,
(CASE WHEN subject = 'Music' THEN score END) as Music
FROM scores
```

I can start to build out my matrix table using a select query.

	name character varying (100)	english numeric	history numeric	math numeric	music numeric
1	Adam	[null]	[null]	10.00	[null]
2	Adam	8.00	[null]	[null]	[null]
3	Adam	[null]	8.00	[null]	[null]
4	Adam	[null]	[null]	[null]	8.00
5	Linda	[null]	[null]	12.00	[null]
6	Linda	10.00	[null]	[null]	[null]
7	Linda	[null]	8.00	[null]	[null]
8	Linda	[null]	[null]	[null]	6.00

I do not want all the NULLs so I need to group by name.

Character: 26

Because I am using group by I must use an aggregate function.

	name character varying (100)	english numeric	history numeric	math numeric	music numeric
1	Linda	10.00	8.00	12.00	6.00
2	Adam	8.00	8.00	10.00	8.00

The final enhancement is to order by name.

Using the case statements I have created a pivot table without using crosstab. I can even order by different columns such as the highest math score.

34.8 Aggregate over a Filter

We can use aggregate and filter function to get our pivot table done too.

```
SELECT
location,
SUM(raindays) FILTER (WHERE year = '2012') as "2012",
SUM(raindays) FILTER (WHERE year = '2013') as "2013",
SUM(raindays) FILTER (WHERE year = '2014') as "2014",
SUM(raindays) FILTER (WHERE year = '2015') as "2015",
SUM(raindays) FILTER (WHERE year = '2016') as "2016",
SUM(raindays) FILTER (WHERE year = '2017') as "2017"
FROM rainfalls
GROUP BY location
ORDER BY 1;
```

I can filter a sum of raindays to only include from specific years. I can order by column 1 or order by 2017 raindays.

```
SELECT
location,
SUM(raindays) FILTER (WHERE year = '2012') as "2012",
SUM(raindays) FILTER (WHERE year = '2013') as "2013",
SUM(raindays) FILTER (WHERE year = '2014') as "2014",
SUM(raindays) FILTER (WHERE year = '2015') as "2015",
SUM(raindays) FILTER (WHERE year = '2016') as "2016",
SUM(raindays) FILTER (WHERE year = '2017') as "2017"
FROM rainfalls
GROUP BY location
ORDER BY 6;
```

	location text	2012 bigint	2013 bigint	2014 bigint	2015 bigint	2016 bigint	2017 bigint
1	Dubai	112	116	111	80	110	102
2	France	114	111	113	85	120	110
3	Germany	153	120	136	128	138	113
4	London	116	138	137	101	117	110
5	Malaysia	53	77	78	50	52	43
6	Qatar	86	116	111	83	102	89
7	Singapore	124	132	142	111	106	110
8	Sydney	98	120	110	93	122	115
9	Tokyo	47	63	68	53	54	43

	location text	2012 bigint	2013 bigint	2014 bigint	2015 bigint	2016 bigint	2017 bigint
1	Malaysia	53	77	78	50	52	43
2	Tokyo	47	63	68	53	54	43
3	Qatar	86	116	111	83	102	89
4	Singapore	124	132	142	111	106	110
5	Dubai	112	116	111	80	110	102
6	London	116	138	137	101	117	110
7	France	114	111	113	85	120	110
8	Sydney	98	120	110	93	122	115
9	Germany	153	120	136	128	138	113

```

SELECT
location,
ROUND(AVG(raindays) FILTER (WHERE year = '2012'),2) as "2012",
ROUND(AVG(raindays) FILTER (WHERE year = '2013'),2) as "2013",
ROUND(AVG(raindays) FILTER (WHERE year = '2014'),2) as "2014",
ROUND(AVG(raindays) FILTER (WHERE year = '2015'),2) as "2015",
ROUND(AVG(raindays) FILTER (WHERE year = '2016'),2) as "2016",
ROUND(AVG(raindays) FILTER (WHERE year = '2017'),2) as "2017"
FROM rainfalls
GROUP BY location
ORDER BY 1;

```

A pivot table can be created with a traditional SQL query or a crosstab query.

	location text 	2012 numeric 	2013 numeric 	2014 numeric 	2015 numeric 	2016 numeric 	2017 numeric 
1	Dubai	9.33	9.67	9.25	6.67	9.17	8.50
2	France	9.50	9.25	9.42	7.08	10.00	9.17
3	Germany	12.75	10.00	11.33	10.67	11.50	9.42
4	London	9.67	11.50	11.42	8.42	9.75	9.17
5	Malaysia	4.42	6.42	6.50	4.17	4.33	3.58
6	Qatar	7.17	9.67	9.25	6.92	8.50	7.42
7	Singapo...	10.33	11.00	11.83	9.25	8.83	9.17
8	Sydney	8.17	10.00	9.17	7.75	10.17	9.58
9	Tokyo	3.92	5.25	5.67	4.42	4.50	3.58

34.9 Static to Dynamic Pivots

1. We have seen in both the crosstab and the traditional queries form have the drawback that:
 - Output columns must be specifically enumerated
 - They must be added manually to the list.
2. These queries also lack flexibility.
 - To change the order of the columns, or transpose a different column of the source data. For instance have cities on the horizontal axis in stead of years, they need to be rewritten.
3. Also some pivots may have hundreds of columns so listing them manually is too tedious.

Dynamic Pivots

4. A polymorphic query that would automatically have rows values transposed into columns without the need to edit the SQL statement.
 - Get column names
 - Build crosstab query
 - Execute crosstab query

34.10 Dynamic Pivot query via JSON

1. The difficulty in creating a dynamic pivot is:

- In an SQL query, the output columns must be determined before execution, but to know which columns are formed from transposing rows, we would need to execute the query first.

2. One solution may be to have pivoted part encapsulated inside a single column with a composite or array type style. We can use JSON to solve this problem.

We will use `json_object_agg` i.e. aggregates name/value pairs as a JSON object.

```
SELECT
    location,
    json_object_agg(year,raindays ORDER BY year) as "mydata"
FROM rainfalls
GROUP BY location;
```

I get an array of year:
raindays value pairs for
each month of the year.

	location text 	mydata json
1	Dubai	{ "2012": 11, "2012": 4, "2012": 5, "2012": 18, "2012": 8, "2012": 15, "2012": 8, "2012": 9, "2012": 8, "2012": 8, "2012": 7, "2012": 11, "2013": 5, "2013": 14, "2013": 10, "
2	France	{ "2012": 7, "2012": 1, "2012": 3, "2012": 14, "2012": 7, "2012": 17, "2012": 9, "2012": 3, "2012": 8, "2012": 16, "2012": 10, "2012": 19, "2013": 4, "2013": 10, "2013": 11,
3	Germany	{ "2012": 24, "2012": 12, "2012": 5, "2012": 4, "2012": 18, "2012": 8, "2012": 16, "2012": 15, "2012": 9, "2012": 6, "2012": 20, "2012": 16, "2013": 15, "2013": 10, "2013": 1,
4	London	{ "2012": 5, "2012": 4, "2012": 18, "2012": 12, "2012": 16, "2012": 6, "2012": 3, "2012": 11, "2012": 8, "2012": 8, "2012": 7, "2012": 18, "2013": 18, "2013": 13, "2013": 1,
5	Malaysia	{ "2012": 10, "2012": 4, "2012": 0, "2012": 2, "2012": 8, "2012": 8, "2012": 1, "2012": 1, "2012": 1, "2012": 6, "2012": 8, "2012": 4, "2013": 2, "2013": 10, "2013": 6, "2013
6	Qatar	{ "2012": 5, "2012": 6, "2012": 12, "2012": 5, "2012": 8, "2012": 5, "2012": 8, "2012": 4, "2012": 8, "2012": 8, "2012": 14, "2012": 3, "2013": 15, "2013": 9, "2013": 15, "20
7	Singapore	{ "2012": 5, "2012": 21, "2012": 10, "2012": 19, "2012": 5, "2012": 6, "2012": 10, "2012": 14, "2012": 7, "2012": 14, "2012": 7, "2012": 6, "2013": 19, "2013": 13, "2013": 1,
8	Sydney	{ "2012": 12, "2012": 14, "2012": 6, "2012": 6, "2012": 6, "2012": 6, "2012": 9, "2012": 15, "2012": 6, "2012": 11, "2012": 3, "2012": 4, "2013": 10, "2013": 11, "2013": 15,
9	Tokyo	{ "2012": 1, "2012": 4, "2012": 2, "2012": 7, "2012": 6, "2012": 6, "2012": 7, "2012": 4, "2012": 2, "2012": 2, "2012": 2, "2012": 4, "2013": 7, "2013": 6, "2013": 2, "2013": 1,

```

SELECT
    location,
    json_object_agg(year,total_raindays ORDER BY year) as "mydata"
FROM
(
    SELECT
        location,
        year,
        SUM(raindays) AS total_raindays
    FROM rainfalls
    GROUP BY
        location,
        year
) s
GROUP BY location;

```

Instead of selecting from the table, I am going to select from a subquery which is doing the aggregation per year.

	location text 	mydata json 
1	Dubai	{ "2012" : 112, "2013" : 116, "2014" : 111, "2015" : 80, "2016" : 110, "2017" : 102 }
2	France	{ "2012" : 114, "2013" : 111, "2014" : 113, "2015" : 85, "2016" : 120, "2017" : 110 }
3	Germany	{ "2012" : 153, "2013" : 120, "2014" : 136, "2015" : 128, "2016" : 138, "2017" : 113 }
4	London	{ "2012" : 116, "2013" : 138, "2014" : 137, "2015" : 101, "2016" : 117, "2017" : 110 }
5	Malaysia	{ "2012" : 53, "2013" : 77, "2014" : 78, "2015" : 50, "2016" : 52, "2017" : 43 }
6	Qatar	{ "2012" : 86, "2013" : 116, "2014" : 111, "2015" : 83, "2016" : 102, "2017" : 89 }
7	Singapore	{ "2012" : 124, "2013" : 132, "2014" : 142, "2015" : 111, "2016" : 106, "2017" : 110 }
8	Sydney	{ "2012" : 98, "2013" : 120, "2014" : 110, "2015" : 93, "2016" : 122, "2017" : 115 }
9	Tokyo	{ "2012" : 47, "2013" : 63, "2014" : 68, "2015" : 53, "2016" : 54, "2017" : 43 }

Now I get a JSON return that is a series of value pairs of total raindays per year.

The output is not as visually satisfying as a table but the data is dynamically prepared without specifying output columns.

34.10 Dynamic Pivot Table Columns

1. Prepare a function which will return the full cross tab query.
2. Using the following syntax
 - Master query **SELECT FROM GROUP BY**
 - Header columns query **SELECT DISTINCT(column_name) FROM**
3. For Dynamic columns we will have them lower case, add a prefix '_' so that to make them unique too.

4. The final output syntax:

```
SELECT FROM crosstab (  
    'master_query',  
    'header_columns_query'  
) as newtable  
(  
    row_column_name VARCHAR,  
    _col1  
    _col2  
    ...  
)
```

The 'pivotcode' function will use the following:

Tablename	name of source table you want to pivot
Myrow	name of the column in source table you want to be the rows
Mycol	name of the column in the source table you want to be the columns
Mycell	An aggregate expression determined how the cells values will be created
Celldatatype	desired datatype for the cells


```
SELECT pivotcode (tablename, myrow, mycol, mycell, celldatatype)
```

To use the function, I just call it with select, dropping in the values for all the variables.

```
SELECT pivotcode ('rainfalls', 'location', 'year', 'sum(raindays)', 'NUMERIC');
```

	pivotcode character varying
1	SELECT * FROM crosstab (

The return is not actually the pivot table but a dynamically built query that I can use to get a pivot table. If I click on the cell I get a pop up with the full query in it which I can then run to get my pivot table table.

```
SELECT * FROM crosstab (
  'SELECT location,year,sum(raindays) FROM rainfalls GROUP BY 1,2 ORDER BY 1,2',
  'SELECT DISTINCT year FROM rainfalls ORDER BY 1'
)
AS newtable (
  location VARCHAR,_2012 NUMERIC,_2013 NUMERIC,_2014 NUMERIC,_2015 NUMERIC,_2016
NUMERIC,_2017 NUMERIC
);
```

In the real world you could call this function then get the custom built SQL query as a response then you would run the SQL query.

	location character varying	_2012 numeric	_2013 numeric	_2014 numeric	_2015 numeric	_2016 numeric	_2017 numeric
1	Dubai	112	116	111	80	110	102
2	France	114	111	113	85	120	110
3	Germany	153	120	136	128	138	113
4	London	116	138	137	101	117	110
5	Malaysia	53	77	78	50	52	43
6	Qatar	86	116	111	83	102	89
7	Singapore	124	132	142	111	106	110
8	Sydney	98	120	110	93	122	115
9	Tokyo	47	63	68	53	54	43

34.11 Interactive Client-side Pivot

1. Pgsql offers a client-side approach for pivots through the command `\crosstabview`
2. In interactive use, this method is probably the quickest way to visualise pivoted representations.
3. From the terminal write the query then ed it with `\crosstabview`

```
movies=# SELECT location, year, SUM(raindays) FROM rainfalls GROUP BY location, year
HAVING SUM(raindays) > 120 ORDER BY location \crosstabview
 location | 2014 | 2016 | 2015 | 2012 | 2013
-----+-----+-----+-----+-----+-----
 Germany  | 136  | 138  | 128  | 153  |
 London   | 137  |      |      |      | 138
 Singapore| 142  |      |      | 124  | 132
 Sydney   |      | 122  |      |      |
(4 rows)
```

4. To have other transposition, we just have to give year and city as arguments to `crosstabview` in that order, without changing the query.

```
movies=# SELECT location, year, SUM(raindays) FROM rainfalls GROUP BY location, year
HAVING SUM(raindays) > 120 ORDER BY location \crosstabview year location
 year | Germany | London | Singapore | Sydney
-----+-----+-----+-----+-----
 2014 | 136     | 137    | 142       |
 2016 | 138     |        |           | 122
 2015 | 128     |        |           |
 2012 | 153     |        | 124       |
 2013 |         | 138    | 132       |
(5 rows)
```

34.12 Handling missing Values

You can also use crosstab in the following syntax too. It produces a 'pivot table' with the value columns specified by a SECOND QUERY.

`crosstab(text source_sql, text category_sql)`

Get all directors first_name with their total revenues per each year, using crosstab.

```
SELECT * FROM crosstab (  
    $$  
    SELECT  
        d.first_name,  
        EXTRACT('year' FROM mv.release_date) as year,  
        SUM(r.revenues_domestic + r.revenues_international)::INT  
    FROM directors d  
    INNER JOIN movies mv ON mv.director_id = d.director_id  
    INNER JOIN movies_revenues r ON r.movie_id = mv.movie_id  
    WHERE mv.release_date BETWEEN '2010-01-01' AND '2020-10-31'  
    GROUP BY 1,2  
    ORDER BY 1,2  
    $$  
) AS ct  
(  
    director VARCHAR,  
    "2010" INT,  
    "2011" INT,  
    "2012" INT,  
    "2013" INT,  
    "2014" INT,  
    "2015" INT,  
    "2016" INT,  
    "2017" INT  
);
```

Typically the second query would be enclosed in single quotes but this would fail because the dates in the between clause contained un-escaped single quotes.

But I can replace the containing single quotes with \$\$ to solve this.

	director character varying 🔒	2010 integer 🔒	2011 integer 🔒	2012 integer 🔒	2013 integer 🔒	2014 integer 🔒	2015 integer 🔒	2016 integer 🔒	2017 integer 🔒
1	Ang	609	[null]	[null]	[null]	[null]	[null]	[null]	[null]
2	Mark	10	[null]	[null]	[null]	[null]	[null]	[null]	[null]
3	Martin	159							
4	Richard	1							
5	Sion	[null]							
6	Wes	175							

But the return of this query is messed up.

It is putting all the values in the first column and not distributing it across the year columns.

Crosstab queries often produce errors like this where the query runs but the data does not look right.

We can fix this with two queries i.e. a source_query and a columns_query

```
SELECT * FROM crosstab (
    $$
    -- source query
    SELECT
        d.first_name,
        EXTRACT('year' FROM mv.release_date) as year,
        SUM(r.revenues_domestic + r.revenues_international)::INT
    FROM directors d
    INNER JOIN movies mv ON mv.director_id = d.director_id
    INNER JOIN movies_revenues r ON r.movie_id = mv.movie_id
    WHERE mv.release_date BETWEEN '2010-01-01' AND '2020-10-31'
    GROUP BY 1,2
    ORDER BY 1,2
    $$,
    $$
    -- column SQL
    VALUES (2010),(2011),(2012),(2013),(2014),(2015),(2016),(2017)
    $$
) AS ct
(
    director VARCHAR,
    "2010" INT,
    "2011" INT,
    "2012" INT,
    "2013" INT,
    "2014" INT,
    "2015" INT,
    "2016" INT,
    "2017" INT
);
```

I add a coma after the first dollar quoted query then add the second query that contains a list of values. This is the preferred method to write clean crosstab queries.

	director character varying 🔒	2010 integer 🔒	2011 integer 🔒	2012 integer 🔒	2013 integer 🔒	2014 integer 🔒	2015 integer 🔒	2016 integer 🔒	2017 integer 🔒
1	Ang	[null]	[null]	609	[null]	[null]	[null]	[null]	[null]
2	Mark	10	[null]	[null]	[null]	[null]	[null]	[null]	[null]
3	Martin	[null]	[null]	[null]	[null]	[null]	[null]	[null]	159
4	Richard	[null]	1	[null]	[null]	[null]	[null]	[null]	[null]
5	Sion	[null]	[null]	[null]	[null]	[null]	[null]	[null]	[null]
6	Wes	[null]	[null]	[null]	[null]	175	[null]	[null]	[null]

Now the data is
plotted in the
correct columns

35. Internationalisation

35.1 Global character Set Support

35.2 Client and Server Encoding

35.1 Global character Set Support

The whole idea is how to make PostgreSQL support global languages data.

1. PostgreSQL allows the storage of text in a variety of character sets (also called encodings)

- Single-byte character sets ISO 8859 series
- Multiple-byte character sets EUC(extended Unix code)
UTF-8
Mule internal code

2. The default character set selected while initialising your PostgreSQL database cluster using initdb.

Initdb -E UTF8

3. You can have multiple databases each with a different character set.

<https://www.postgresql.org/docs/current/multibyte.html>

4. To create a UTF-8 based database

```
CREATE DATABASE database_utf8
WITH OWNER "postgres"
ENCODING 'UTF8'
LC_COLLATE = 'en_GB.utf8'
LC_CTYPE = 'en_GB.utf8'
TEMPLATE template0;
```

UTF_8 can support all world languages and contain data in multiple languages.

5. To create a KOREAN based language database with character set EUC_KR

```
CREATE DATABASE database_euc_kr
WITH OWNER "postgres"
ENCODING 'EUC_KR'
LC_COLLATE = 'ko_KR.euckr'
LC_CTYPE = 'ko_KR.euckr'
TEMPLATE template0;
```

movies

General Definition Security Parameters Default Privileges Advanced SQL

Encoding UTF8

Tablespace pg_default

Locale Provider libc

Collation English_United Kingdom.1252

Character type English_United Kingdom.1252

ICU Locale

ICU Rules

Builtin Locale

Connection limit -1

Template? ☐

Note: When the preferences setting 'show template databases'

Close Reset Save

```
CREATE DATABASE database_utf8
WITH OWNER "postgres"
ENCODING 'UTF8'
LC_COLLATE = 'en_GB.utf8'
LC_CTYPE = 'en_GB.utf8'
TEMPLATE template0;
```

If I right click on the database → properties. Under the Definition tab I can see the language of the database and the encoding.

6. PostgreSQL will allow superusers to create databases with SQL_ASCII encoding even when LC_CTYPE is not C or POSIX. As noted above, SQL_ASCII does not enforce that the data stored in the database has any particular encoding. It is not a good choice to have encoding set to SQL_ASCII

7. PostgreSQL can determine which character set is applied by the _LC_CTYPE setting, and it will enforce that only the matching database encoding is used.

35.1 Client and Server Encoding

The first notion to understand when processing text in any program is of course encoding. For example the letter A is 7-bits byte 1000001, or 65 in decimal or 41 in hexadecimal.

Server encoding is largely geared towards the server and what is the encoding used to add the data inside the database.

```
SHOW SERVER_ENCODING;
```

server_encoding text	
1	UTF8

Client encoding is to interact with the database and how to read the data it gets from the db.

```
SHOW CLIENT_ENCODING;
```

client_encoding text	
1	UTF8

It is highly recommended to keep the SERVER_ENCODING and the CLIENT_SIDE encoding the same to ensure that they can both understand the data being passed.

```
SET CLIENT_ENCODING TO UTF8;  
SET  
Query returned successfully in 51 msec.
```

```
CREATE TABLE table_encoding(
  lang_id SERIAL PRIMARY KEY,
  lang_code TEXT,
  lang_text TEXT
);
```

```
SELECT * FROM table_encoding;
```

	lang_id [PK] integer	lang_code text	lang_text text
1	1	English	Hello
2	2	Arabic	مرحبًا
3	3	Georgian	გამარჯობა

```
INSERT INTO table_encoding (lang_code, lang_text)
VALUES ('English', 'Hello');
```

```
INSERT INTO table_encoding (lang_code, lang_text)
VALUES ('Arabic', 'مرحبًا');
```

```
INSERT INTO table_encoding (lang_code, lang_text)
VALUES ('Georgian', 'გამარჯობა');
```

I can insert words from any world language into UTF8.
Note that the database is set to UTF8 server encoding and client encoding.

```
SHOW SERVER_ENCODING;
```

	server_encoding text
1	UTF8

```
SHOW CLIENT_ENCODING;
```

	client_encoding text
1	UTF8

```
SELECT * FROM table_encoding WHERE lang_code ~ 'Georgian';
```

	lang_id [PK] integer	lang_code text	lang_text text
1	3	Georgian	გამარჯობა

```
SET CLIENT_ENCODING TO latin1;
```

```
SELECT * FROM table_encoding WHERE lang_code ~ 'Georgian';
ERROR: character with byte sequence 0xe1 0x83 0x92 in
encoding "UTF8" has no equivalent in encoding "LATIN1"
SQL state: 22P05
```

```
SHOW CLIENT_ENCODING;
```

	client_encoding text
1	LATIN1

```
SET CLIENT_ENCODING TO UTF8;
```

The client and server encoding must match.

36. PostgreSQL Transactions

[36.1 What is a transaction?](#)

[36.2 How SQL Protects Database During Transaction](#)

[36.3 ACID Database](#)

[36.4 Transaction Setup](#)

[36.5 Transaction Analysis](#)

[36.6 How to Fix Aborted Transaction](#)

[36.7 How to Fix Transaction on Crash](#)

[36.8 Partial Transaction rollback with Savepoints](#)

[36.9 Using SAVEPOINT with Transaction](#)

1. Our database is most vulnerable to damages while we or someone else is CHANGING IT.
2. If a software or hardware failure occurs while the change is in progress, a database may be left in a indeterminate state...
3. A transaction is a list of commands that we want to run on our database to make changes on the information.
4. A transaction bundles multiple steps or operations into a single, ALL-OR-NOTHING operation.
5. SQL protects our database by restricting operations that can change it, so they can occur only WITHIN-TRANSACTIONS.
6. There are four main transaction commands, that protect from hard, both accidental and intentional.
 - COMMIT
 - ROLLBACK
 - GRANT
 - REVOKE
7. It is possible for a hardware or software problem to occur and as a result your database is susceptible to damage.
8. To minimise the chances of damage, the DBMS closes the window of vulnerability as much as possible by performing all OPERATIONS THAT AFFECT THE DATABASE WITHIN A TRANSACTION and then commit all these changes/operations at once, AT THE END OF THE TRANSACTION.
9. In a multi-user system, database corruption or incorrect results are possible even if no hardware or software failures occur. Interactions between two or more users who access the same table at the same time can cause serious data issues and more. By restricting changes so that they occur only WITHIN TRANSACTIONS, SQL addresses these problems accordingly.

36.2 How SQL Protects Database During Transaction

1. During a transaction, SQL records every operation performed on the data in a log file.
2. If anything interrupts the transaction BEFORE the COMMIT statement ends the transaction, you can restore the system to its original state by issuing a ROLLBACK statement. For example if a transaction is BEGIN → INSERT → UPDATE → CRASH → COMMIT then at the point of crash the commit would not be performed and the insert and update discarded. A rollback could be performed because it occurred before the commit.
3. The ROLLBACK processes the transaction log in REVERSE, undoing all the actions that took place in the transaction.

36.3 ACID Database

1. Database designers often use the term that "they want their database to have ACID"
2. Acid is simply an Acronym for: **A**TOMICITY, **C**ONSISTENCY, **I**SOLATION, **D**URABILITY.

All of the above four characteristics are necessary to protect our database from corruption. It is like ALL OR NOTHING.....

Atomicity

Database transactions should be 'atomic' in nature i.e.

- The entire transaction should be treated as an INDIVIDUAL UNIT.
- Either it is executed in this entirety (COMMIT), or the database is restored (ROLLBACK) to it's original state before the transaction is executed.

Consistency

The meaning of consistency here is not consistent: It should vary from application to application.

- For example, in a bank application or environment, when you transfer funds from one account to another, you want the total amount of money from both account (from/to accounts) to be the SAME as at the END OF THE TRANSACTION. Over here consistency means more of a BALANCED APPROACH before and after a transaction is executed!

Isolation

Database transactions should be totally isolated from other transactions that execute at the SAME TIME.

- If the transactions are SERIALISED then only TOTAL ISOLATION is achieved.

Durability

After a transaction has committed or rolled back, we should be able to count on the database BEING IN THE PROPER STATE, i.e. UP TO DATE.

- Even if your system suffers a hard crash or any software downturn after a COMMIT, but before the transaction is stored in disk, a well designed and durable DBMS should guarantee that UPON RECOVERY from the crash, the database can be restored to it's PROPER STATE!

36.4 Transaction Setup

```
CREATE TABLE accounts (  
    account_id SERIAL PRIMARY KEY,  
    name VARCHAR(100) NOT NULL,  
    balance INTEGER NOT NULL  
);
```

To demonstrate how transactions occur in PostgreSQL, we will setup a table and add some data into it.

```
INSERT INTO accounts (name, balance) VALUES  
( 'Adam', 100),  
( 'Bob', 100),  
( 'Linda', 100);
```

```
SELECT * FROM  
accounts;
```

	account_id [PK] integer	name character varying (100)	balance integer
1	1	Adam	100
2	2	Bob	100
3	3	Linda	100

36.5 Transaction Analysis

Connect to our PostgreSQL database via a connection. i.e. this query editor is connection #1

Note how if I hover over the connection icon it says "The session is idle and there is no current Transaction". The reason is that behind the scene what PostgreSQL is doing is adding a begin and a Commit.

The screenshot shows a PostgreSQL query editor interface. At the top, there are two tabs: "movies/postgres@PostgreSQL 17*" and "movies/postgres@...". The first tab is active. Below the tabs, there is a status bar that says "The session is idle and there is no current transaction." Below the status bar, there is a "Query" tab and a "Query History" tab. The "Query" tab is active, and it shows the following SQL queries:

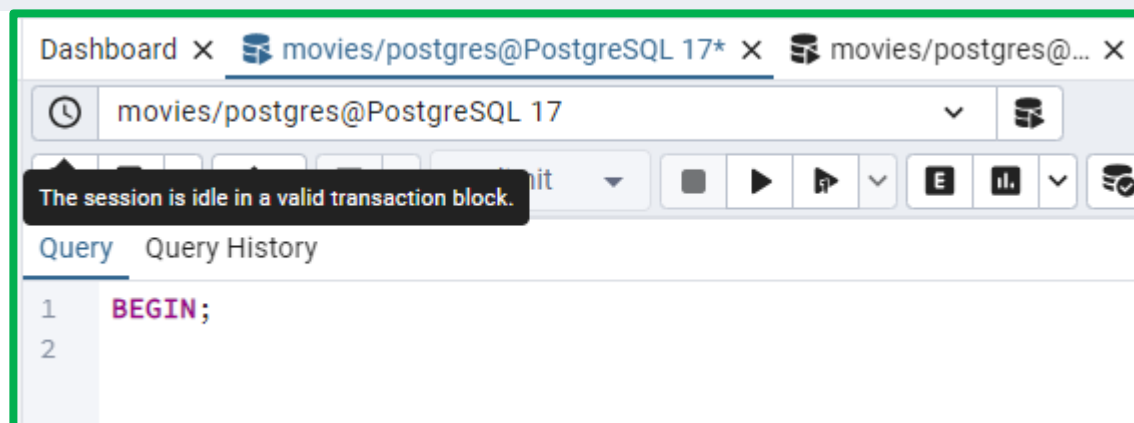
```
3      name VARCHAR(100) NOT NULL,  
4      balance INTEGER NOT NULL  
5  );  
6  
7  INSERT INTO accounts  
8  ('Adam', 100),  
9  ('Bob', 100),  
10 ('Linda', 100);
```

Overlaid on the bottom right of the screenshot is a black box containing the following SQL queries:

```
BEGIN;  
SELECT * FROM accounts;  
COMMIT;
```



Note that when I type Begin; the icon changes to a clock and it says "The session is idle in a valid transaction block"



```
BEGIN;
UPDATE accounts
SET balance = balance - 50
WHERE name = 'adam';
WARNING: there is already a transaction in progress
UPDATE 0

Query returned successfully in 42 msec.
```

In connection #1 I update a row but it gives me a warning saying there is already a transaction in progress and that no rows have been updated.

```
BEGIN;
UPDATE accounts
SET balance = balance - 50
WHERE name = 'adam';
COMMIT;
WARNING: there is already a transaction in progress
COMMIT

Query returned successfully in 39 msec.
```

If I add a commit the the transaction then it still says there is a transaction in progress.

```
BEGIN;
UPDATE accounts
SET balance = balance - 50
WHERE name = 'adam';
COMMIT;
WARNING: there is no transaction in progress
COMMIT

Query returned successfully in 39 msec.
```

The only way to solve this is to perform a rollback.

```
SELECT * FROM accounts;
```

Note that in the second connection window #2 the update has not been performed. Because I hung up the first connection and had to rollback.

	account_id [PK] integer	name character varying (100)	balance integer
1	1	Adam	100
2	2	Bob	100
3	3	Linda	100

```
BEGIN;  
UPDATE accounts  
SET balance = balance - 50  
WHERE name = 'Adam';  
COMMIT;  
COMMIT
```

Query returned successfully in 54 msec.

Now I see the update has actually been performed in the data.

I connect to the database via another connection. i.e, by launching another query editor - connection #2.

```
SELECT * FROM accounts;
```

	account_id [PK] integer	name character varying (100)	balance integer
1	2	Bob	100
2	3	Linda	100
3	1	Adam	50

Conclusions:

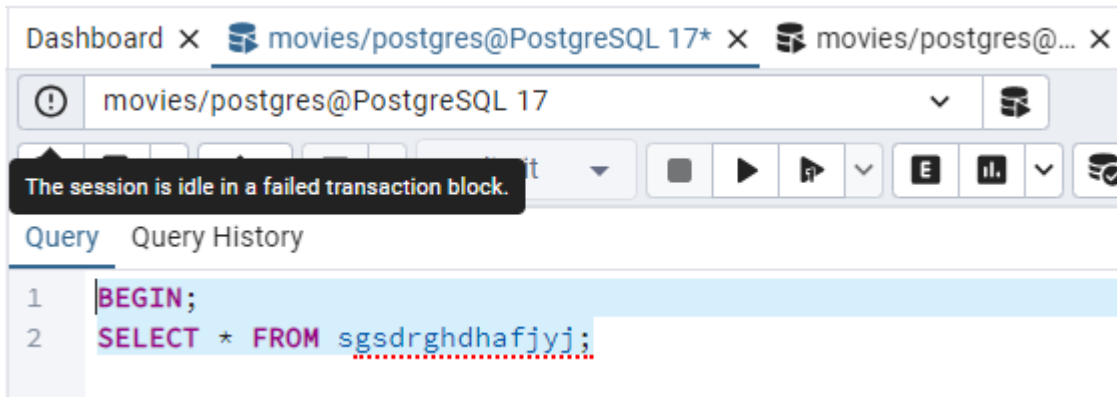
Changes to data in the database are performed within a Transaction which starts with a begin. Only when the commit is performed do the changes take effect. If there is a problem during the transaction, before the commit then the changes are not made to the data.

36.6 How to Fix Aborted Transaction

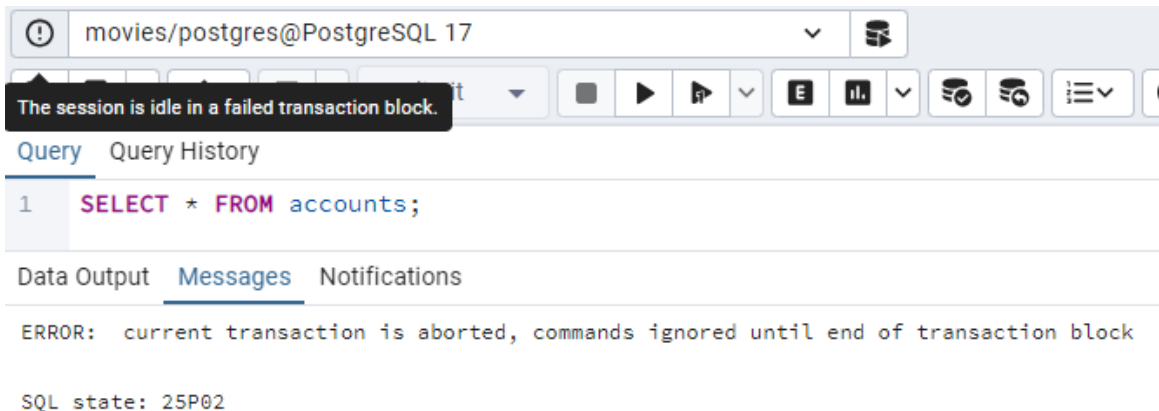
```
SELECT * FROM sgdrghdhafjyj;  
ERROR: relation "sgdrghdhafjyj" does not exist  
LINE 1: SELECT * FROM sgdrghdhafjyj;  
                        ^  
  
SQL state: 42P01  
Character: 15  
  
SELECT * FROM accounts;
```

If I run a `SELECT * FROM bogus_table_name` and get an error then I run a `SELECT * FROM genuine_table_name` it works?

It is because each statement is implicitly wrapped in a transaction block of `Begin` → `Commit`.



Now if I run the same but with `begin` to start a transaction. Note that the connection is in a failed transaction block.



Now because I am in a failed transaction state I cannot perform any other statements.

movies/postgres@PostgreSQL 17

The session is idle and there is no current transaction.

Query Query History

```
1 ROLLBACK;
```

Data Output Messages Notifications

ROLLBACK

Query returned successfully in 98 msec.

However if I issue a ROLLBACK, it will allow me to fix the connection state to idle, no current transaction.

movies/postgres@PostgreSQL 17

No limit

Query Query History

```
1 SELECT * FROM accounts;
```

Data Output Messages Notifications

	account_id [PK] integer	name character varying (100)	balance integer
1	2	Bob	100
2	3	Linda	100
3	1	Adam	50

Now I can use the connection again.

The roll back kills the failing connection by undoing the previous action.

```
UPDATE accounts  
SET balance = 200;  
  
SELECT * FROM accounts;
```

	account_id [PK] integer	name character varying (100)	balance integer
1	2	Bob	200
2	3	Linda	200
3	1	Adam	200

36.7 How to Fix Transaction on Crash

Now I begin a transaction block and make some changes and I can see within this transaction the changes but the change has not been committed so the data has not been written to disk.

```
BEGIN;  
UPDATE accounts  
SET balance = 50  
WHERE name = 'Adam';  
UPDATE 1  
Query returned successfully in 55 msec.
```

```
SELECT * FROM accounts;
```

	account_id [PK] integer	name character varying (100)	balance integer
1	2	Bob	200
2	3	Linda	200
3	1	Adam	50

Dashboard x movies/postgres@... x movies/postgres@... x

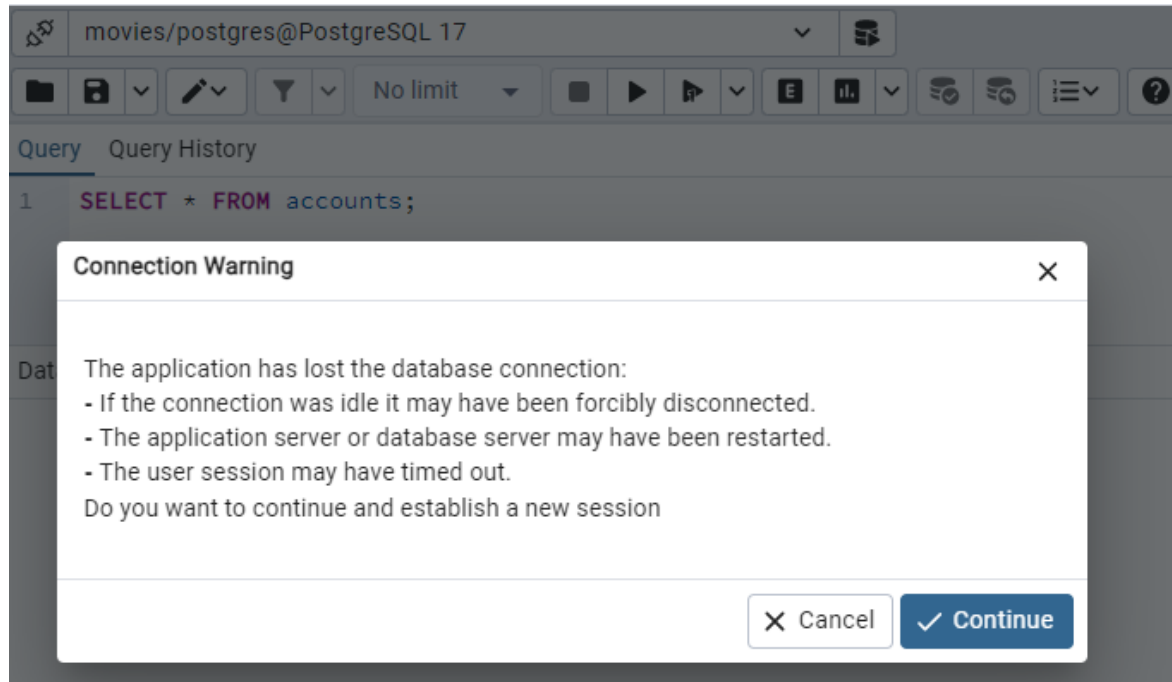
Activity State Configuration Logs System

Sessions

☐ Active sessions only

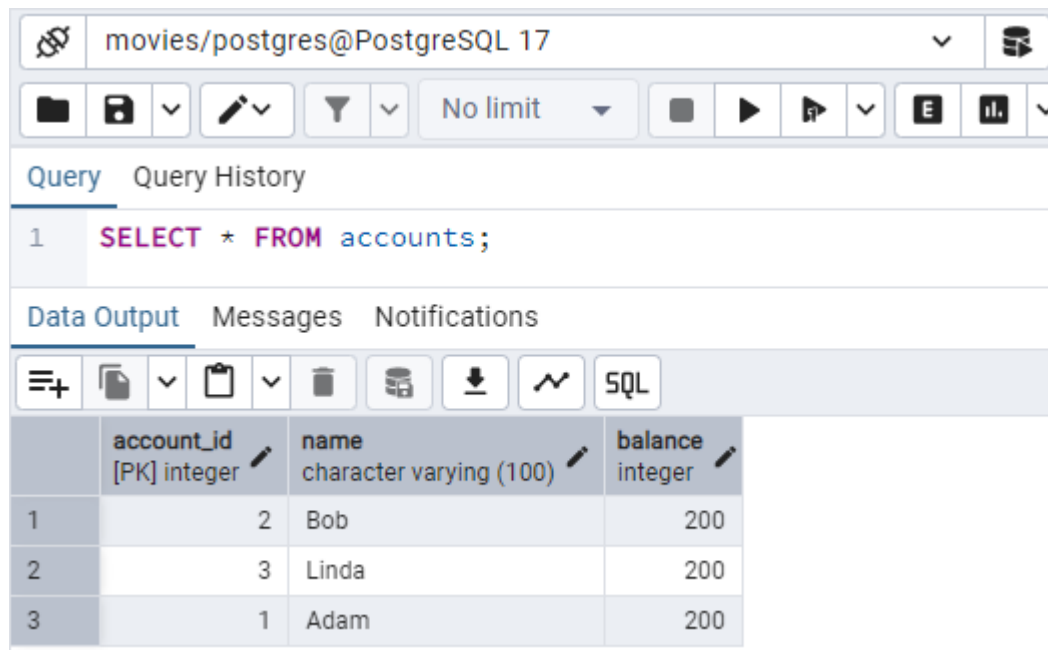
			PID	User	Application	Client	Backend start	Transaction start	State	Wait event
✗	■	>	772	postgr...	pgAdmin 4 - CONN:5...	::1	2025-02-21 21:21:11...		idle	Client: ClientRead
✗	■	>	3140	postgr...	pgAdmin 4 - CONN:3...	::1	2025-02-21 21:21:08...	2025-02-21 22:19:42...	idle i...	Client: ClientRead
✗	■	>	11832	postgr...	pgAdmin 4 - DB:mov...	::1	2025-02-21 15:14:41...	2025-02-21 22:25:33...	active	

From the Dashboard of PgAdmin4 I can terminate the two sessions.



Now if I try and run a statement in the connection, I get a message asking me if I would like to establish a new session.

I click continue.



And the output is that the changes were not made.

If a connection is lost before the commit PostgreSQL will automatically perform the Rollback.

This is due to ACID design where all transactions are atomic, Consistent, isolated and Durable.

36.8 Partial Transaction Rollback with Savepoints

1. Simple ROLLBACK and COMMIT statements enable us to write or undo an ENTIRE TRANSACTION. However we might want sometimes a support for a ROLLBACK of a PARTIAL TRANSACTION.
2. To support the ROLLBACK of PARTIAL TRANSACTION, we must out PLACEHOLDERS at strategic locations of the transaction block, Thus, if a rollback is required, you can read back on the said placeholder.
3. In PostGreSQL these placeholders are called 'savepoints'
4. Syntax: SAVE savepoint_name;
5. Each Savepoint takes a unique name that identifies it so that when you roll back, the DBMS knows where you are rolling back to.
6. To rollback a transaction to a savepoint, use the syntax: ROLLBACK TO savepoint_name.
7. So full building block with transaction will look like:

BEGIN;

<your_valid_statement>

SAVEPOINT savepoint1;

<your_valid_statement>

ROLLBACK TO savepoint1;

COMMIT;

36.9 Using SAVEPOINT with Transaction

```
SELECT * FROM accounts;

BEGIN;
UPDATE accounts
SET balance = -50
WHERE name = 'Adam';
SELECT * FROM accounts;
SAVEPOINT first_update;

UPDATE accounts
SET balance = -50
WHERE name = 'Adam';
SELECT * FROM accounts;
```

I am doing a couple of update queries within a transaction block. However, after the second query statement, I notice that for some reason I don't want that update. Note how after the first update there is a savepoint.

	account_id [PK] integer	name character varying (100)	balance integer
1	2	Bob	200
2	3	Linda	200
3	1	Adam	-50

```
ROLLBACK TO first_update;
ROLLBACK
```

```
Query returned successfully in 53 msec.
```

```
COMMIT;
```

Now I rollback and then commit.

I can have as many savepoints in a transaction as I want. A good naming convention is essential as always. i.e. SP_table_name_changes_description

37. Using Northwind Database

[37.1 Orders Shipping to USA or France](#)

[37.2 Total Number of Orders Shipped to USA or France](#)

[37.3 Orders Shipped to Latin America](#)

[37.4 Show Total Order Amount for each Order](#)

[37.5 First the Oldest and latest Order Date](#)

[37.6 Total Products in each Category](#)

[37.7 List Products in each Category](#)

[37.8 List Products that Need Reordering](#)

[37.9 Freight Analysis](#)

[37.10 Customers with No Orders](#)

[37.11 Top Customers with Total Orders Amount](#)

[37.12 Orders with Many lines of Ordered Items](#)

[37.13 Orders with Double Entry Items](#)

[37.14 Late Shipped Orders by Employees](#)

[37.15 Countries with Customers or Suppliers](#)

[37.16 Countries with Customers or Suppliers – Using CTE](#)

[37.17 Customers with Multiple Orders](#)

[37.18 First Order From each Country](#)

37.1 Orders Shipping to USA or France

```
SELECT
* FROM orders
WHERE ship_country = 'USA'
OR ship_country = 'France'
ORDER BY ship_country;
```

or

```
SELECT
* FROM orders
WHERE ship_country
IN('USA', 'France')
ORDER BY ship_country;
```

Showing rows: 1 to 199 Page No: 1 of 1									
	order_id [PK] smallint	customer_id character	employee_id smallint	order_date date	required_date date	shipped_date date	ship_via smallint	freight real	ship_name character varying (40)
1	10248	VINET	5	1996-07-04	1996-08-01	1996-07-16	3	32.38	Vins et alcools Chevalier
2	10251	VICTE	3	1996-07-08	1996-08-05	1996-07-15	1	41.34	Victuailles en stock
3	10265	BLONP	2	1996-07-25	1996-08-22	1996-08-12	1	55.28	Blondel père et fils
4	10274	VINET	6	1996-08-06	1996-09-03	1996-08-16	1	6.01	Vins et alcools Chevalier
5	10295	VINET	2	1996-09-02	1996-09-30	1996-09-10	2	1.15	Vins et alcools Chevalier
6	10297	BLONP	5	1996-09-04	1996-10-16	1996-09-10	2	5.74	Blondel père et fils
7	10311	DUMON	1	1996-09-20	1996-10-04	1996-09-26	3	24.69	Du monde entier
8	10331	BLONP	4	1996-10-16	1996-11-27	1996-10-21	1	10.10	Rouge ann'

37.1 Total Number of Orders Shipped to USA or France

```
SELECT
    ship_country,
    count(*)
FROM orders
WHERE ship_country
IN('USA', 'France')
GROUP BY ship_country
ORDER BY 2 DESC;
```

	ship_country character varying (15)	count bigint
1	USA	122
2	France	77

37.2 Orders Shipped to Latin America

```
SELECT
*
FROM orders
WHERE ship_country
IN('Brazil', 'Mexico', '
Argentina', 'Venezuela'
)
ORDER BY ship_country;
```

Showing rows: 1 to 173 Page No: 1 of 1						
	name character varying (40)	ship_address character varying (60)	ship_city character varying (15)	ship_region character varying (15)	ship_postal_code character varying (10)	ship_country character varying (15)
13	no Atlántico Ltda.	Ing. Gustavo Moncada 8585 Piso 20-A	Buenos Aires	[null]	1010	Argentina
14	no grande	Av. del Libertador 900	Buenos Aires	[null]	1010	Argentina
15	no Atlántico Ltda.	Ing. Gustavo Moncada 8585 Piso 20-A	Buenos Aires	[null]	1010	Argentina
16	is Comidas para llevar	Cerrito 333	Buenos Aires	[null]	1010	Argentina
17	ri Carnes	Rua do Paço 67	Rio de Janeiro	R.I	05454-876	Brazil

37.4 Show Total Order Amount for each Order

```
SELECT
  order_id,
  product_id,
  unit_price,
  quantity,
  discount,
  (SUM(unit_price * quantity) - discount) as total_amount
FROM order_details;
```

Showing rows: 1 to 1000						
	order_id [PK] smallint	product_id [PK] smallint	unit_price real	quantity smallint	discount real	total_amount double precision
1	10981	38	263.5	60	0	15810
2	10865	38	263.5	60	0.05	15809.949999999255
3	10417	38	210.8	50	0	10540.00015258789
4	10889	38	263.5	40	0	10540
5	10353	38	210.8	50	0.2	10539.80015258491
6	10424	38	210.8	49	0.2	10329.000149533153
7	10897	29	123.79	80	0	9903.200073242188

37.5 First the Oldest and Latest Order Date

```
SELECT
  MIN(order_date) as first_order,
  MAX(order_date) as last_order
FROM orders;
```

	first_order date	last_order date
1	1996-07-04	1998-05-06

37.6 Total Products in each Category

```
SELECT
  category_id,
  count(*) as total_per_catagory
FROM products
GROUP BY category_id
ORDER BY 1;
```

	category_id smallint	total_per_catagory bigint
1	1	12
2	2	12
3	3	13
4	4	10
5	5	7
6	6	6
7	7	5
8	8	12

```
SELECT
  c.category_name,
  count(*) as total_products
FROM products p
INNER JOIN categories c ON
  c.category_id = p.category_id
GROUP BY 1
ORDER BY 2 DESC;
```

	category_name character varying (15)	total_products bigint
1	Confections	13
2	Condiments	12
3	Beverages	12
4	Seafood	12
5	Dairy Products	10
6	Grains/Cereals	7
7	Meat/Poultry	6
8	Produce	5

37.7 List Products that need Reordering

```
SELECT
    product_id,
    product_name,
    units_in_stock,
    reorder_level
FROM products
WHERE
    units_in_stock <= reorder_level
ORDER BY 4 DESC;
```

	product_id [PK] smallint	product_name character varying (40)	units_in_stock smallint	reorder_level smallint
1	64	Wimmers gute Semmelk...	22	30
2	56	Gnocchi di nonna Alice	21	30
3	70	Outback Lager	15	30
4	11	Queso Cabrales	22	30
5	2	Chang	17	25
6	3	Aniseed Syrup	13	25
7	88	Mascarpone Fabbri	9	25

37.8 Freight Analysis

```
SELECT
    ship_country,
    AVG(freight)
FROM orders
GROUP BY ship_country
ORDER BY 2 DESC
LIMIT 5;
```

	ship_country character varying (15)	avg double precision
1	Austria	184.78750057220458
2	Ireland	145.01262895684494
3	USA	112.87942660355665
4	Germany	92.48590173999794
5	Sweden	87.50270293210004

Top 5 freight
countries


```

SELECT
    ship_country,
    AVG(freight)
FROM orders
WHERE
    order_date BETWEEN ('1997-01-01') AND ('1997-12-31')
GROUP BY ship_country
ORDER BY 2 DESC
LIMIT 5;

```

	ship_country character varying (15) 🔒	avg double precision 🔒
1	Austria	178.36428569612048
2	Switzerland	117.17749977111816
3	Sweden	105.1599993986242
4	Canada	99.2799991404309
5	Ireland	98.07699909210206

Top 5 highest freight charges in 1997

```

SELECT
    ship_country,
    AVG(freight)
FROM orders
WHERE
    EXTRACT('Y' FROM order_date) >= EXTRACT('Y' FROM (SELECT MAX(order_date) FROM orders))
GROUP BY ship_country
ORDER BY 2 DESC
LIMIT 5;

```

	ship_country character varying (15) 🔒	avg double precision 🔒
1	Ireland	339.42248916625977
2	Austria	217.34182132374156
3	USA	153.3223079626377
4	Portugal	102.47000277042389
5	Germany	90.98794135451317

Top 5 highest freight charges last year

37.10 Customers with No Orders

```
SELECT
    c.company_name,
    count(o.orders_id)
FROM customers c
LEFT JOIN orders o ON customer_id = customer_id
WHERE count(o.orders_id) = NULL
ORDER BY 1;
```

	customer_id character	company_name character varying (40)	order_date date
1	FISSA	FISSA Fabrica Inter. Salchichas S.A.	[null]
2	PARIS	Paris spécialités	[null]

37.11 Top Customers with Total Orders Amount

```
SELECT
    c.customer_id,
    c.company_name,
    SUM((od.unit_price * od.quantity) -
    od.discount) as total_amount
FROM customers c
JOIN orders o ON o.customer_id =
c.customer_id
JOIN order_details od ON o.order_id =
o.order_id
GROUP BY c.customer_id, c.company_name
ORDER BY 3 DESC
LIMIT 10;
```

	customer_id [PK] character	company_name character varying (40)	total_amount double precision
1	SAVEA	Save-a-lot Markets	41984464.06362539
2	ERNSH	Ernst Handel	40630126.513185434
3	QUICK	QUICK-Stop	37921451.41230553
4	HUNGO	Hungry Owl All-Night Grocers	25732413.458349027
5	FOLKO	Folk och få HB	25732413.458349027
6	HILAA	HILARION-Abastos	24378075.907909602
7	BERGS	Berglunds snabbköp	24378075.907909602
8	RATTC	Rattlesnake Canyon Grocery	24378075.907909602
9	BONAP	Bon app'	23023738.357470177
10	FRANK	Frankenversand	20315063.256591327

37.12 Orders with Many lines of Ordered Items

```
SELECT
  order_id,
  COUNT(order_id)
FROM order_details
GROUP BY order_id
ORDER BY 2 DESC
LIMIT 5;
```

	order_id smallint 🔒	count bigint 🔒
1	11077	25
2	10979	6
3	10847	6
4	10657	6
5	10514	5

37.13 Orders with Double Entry Items

```
SELECT
  order_id,
  quantity
FROM order_details
WHERE quantity > 60
GROUP BY
  order_id,
  quantity
HAVING COUNT(*) > 1;
```

	order_id smallint 🔒	quantity smallint 🔒
1	10658	70
2	10990	65
3	11030	100

```
WITH duplicate_entries AS
(
  SELECT
    order_id,
    quantity
  FROM order_details
  WHERE quantity > 60
  GROUP BY
    order_id,
    quantity
  HAVING COUNT(*) > 1
)
SELECT *
FROM order_details
WHERE order_id IN (SELECT order_id FROM duplicate_entries)
ORDER BY order_id;
```

	order_id [PK] smallint	product_id [PK] smallint	unit_price real	quantity smallint	discount real
1	10658	21	10	60	0
2	10658	40	18.4	70	0.05
3	10658	60	34	55	0.05
4	10658	77	13	70	0.05
5	10990	21	10	65	0
6	10990	34	14	60	0.15
7	10990	55	24	65	0.15
8	10990	61	28.5	66	0.15
9	11030	2	19	100	0.25
10	11030	5	21.35	70	0
11	11030	29	123.79	60	0.25
12	11030	59	55	100	0.25

37.14 Late Shipped Orders by Employees

```
SELECT
*
FROM orders
WHERE shipped_date > required_date;
```

```
WITH late_orders AS
(
    SELECT
        employee_id,
        count(*) AS total_late_orders
    FROM orders
    WHERE shipped_date > required_date
    GROUP BY employee_id
),
all_orders AS
(
    SELECT
        employee_id,
        COUNT(*) AS total_orders
    FROM orders
    GROUP BY
        employee_id
)
SELECT
    employees.first_name,
    employees.employee_id,
    all_orders.total_orders,
    late_orders.total_late_orders
FROM employees
JOIN all_orders ON all_orders.employee_id = employees.employee_id
JOIN late_orders ON late_orders.employee_id = employees.employee_id
ORDER BY 4 DESC;
```

Data Output Messages Notifications						
						Showing rows: 1 to 37 
	order_id [PK] smallint	customer_id character	employee_id smallint	order_date date	required_date date	shipped_date date
1	10264	FOLKO	6	1996-07-24	1996-08-21	1996-08-23
2	10271	SPLIR	6	1996-08-01	1996-08-29	1996-08-30
3	10280	RFRSS	7	1996-08-14	1996-09-11	1996-09-12

	first_name character varying (10)	employee_id [PK] smallint	total_orders bigint	total_late_orders bigint
1	Margaret	4	156	10
2	Janet	3	127	5
3	Laura	8	104	4
4	Anne	9	43	4
5	Andrew	2	96	4
6	Robert	7	72	4
7	Michael	6	67	3
8	Nancy	1	123	2
9	Steven	5	42	1

37.15 Countries with Customers or Suppliers

```
SELECT
    DISTINCT country
FROM
    customers
UNION ALL
SELECT
    DISTINCT country
FROM
    suppliers
ORDER BY country
```

Data Output Messages Nc

Showing rows: 1 to 37

	country character varying (15)
1	Argentina
2	Australia
3	Austria

37.16 Countries with Customers or Suppliers – Using CTE

```
WITH countries_suppliers AS
(
    SELECT DISTINCT country FROM suppliers
),
countries_customers AS
(
    SELECT DISTINCT country FROM customers
)
SELECT
    countries_suppliers.country as country_suppliers,
    countries_customers.country as country_customers
FROM countries_suppliers
FULL JOIN countries_customers ON countries_customers.country = countries_suppliers.country
ORDER BY 1;
```

Showing rows: 1 to 25 Page No: 1

	country_suppliers character varying (15)	country_customers character varying (15)
1	Australia	[null]
2	Brazil	Brazil
3	Canada	Canada
4	Denmark	Denmark
5	Finland	Finland
6	France	France
7	Germany	Germany

37.17 Customers with Multiple Orders

```
WITH next_order_date AS
(
    SELECT
        customer_id,
        order_date,
        LEAD(order_date, 1) OVER (PARTITION BY customer_id ORDER BY customer_id, order_id)
    AS next_order_date
    FROM orders
)
SELECT
    customer_id,
    order_date,
    next_order_date,
    (next_order_date - order_date) AS days_between_orders
FROM next_order_date
WHERE (next_order_date - order_date) <= 4;
```

Find customers with multiple orders in a 4 day period

Showing rows: 1 to 61  Page No: 1 of 1 				
	customer_id character	order_date date	next_order_date date	days_between_orders integer
1	ANTON	1997-09-22	1997-09-25	3
2	AROUT	1997-11-14	1997-11-17	3
3	BERGS	1996-08-12	1996-08-14	2
4	BERGS	1997-02-12	1997-02-13	1
5	BERGS	1998-02-03	1998-02-06	3
6	BONAP	1997-11-05	1997-11-06	1
7	BONAP	1998-02-05	1998-02-09	4
8	BOTTM	1997-01-10	1997-01-10	0

37.18 First Order From each Country

```
WITH orders_by_country AS
(
    SELECT
        ship_country,
        order_id,
        order_date,
        ROW_NUMBER() OVER (PARTITION BY ship_country ORDER BY ship_country, order_date) as
country_row_number
    FROM orders
)
SELECT
    ship_country,
    order_id,
    order_date
FROM orders_by_country
WHERE country_row_number = 1;
```

Showing rows: 1 to 21  Page No: 1 of 1

	ship_country character varying (15) 	order_id [PK] smallint 	order_date date 
1	Argentina	10409	1997-01-09
2	Austria	10258	1996-07-17
3	Belgium	10252	1996-07-09
4	Brazil	10250	1996-07-08
5	Canada	10332	1996-10-17
6	Denmark	10341	1996-10-29
7	Finland	10266	1996-07-26
8	France	10248	1996-07-04

38. Using HR Database

[38.1 Human Resources Database Structure](#)

[38.2 Quick Overview of All Tables](#)

[38.3 Get Highest, Lowest, Total & average salaries of Employees](#)

[38.4 Difference between Highest & Lowest salaries by job_id](#)

[38.5 Get lowest paid salaries by each Manager](#)

[38.6 Average Salary for each Department with more than 10 employees](#)

[38.7 Average Salary for each Department excluding Programmers](#)

[38.8 Max Salary for each post where salary is equal or above \\$5000](#)

[38.9 Using Alias name with Columns](#)

[38.10 Compute 15% of salaries for all employees](#)

[38.11 List all employee ids with each job id group](#)

[38.12 Discard Characters from Employee Email Address](#)

[38.13 List all Employees with First name beginning with A, B or M](#)

[38.14 String Manipulation with Upper, Lower & initcap functions](#)

[38.15 Using Substring to get Portion of string data](#)

[38.16 Get Unique Designations in Employees table](#)

[38.17 Get Employees with particular department id](#)

[38.18 Select Employees not in Range](#)

[38.19 Find 'c' in last name at 3rd or greater position](#)

[38.20 Update Phone Number with strings](#)

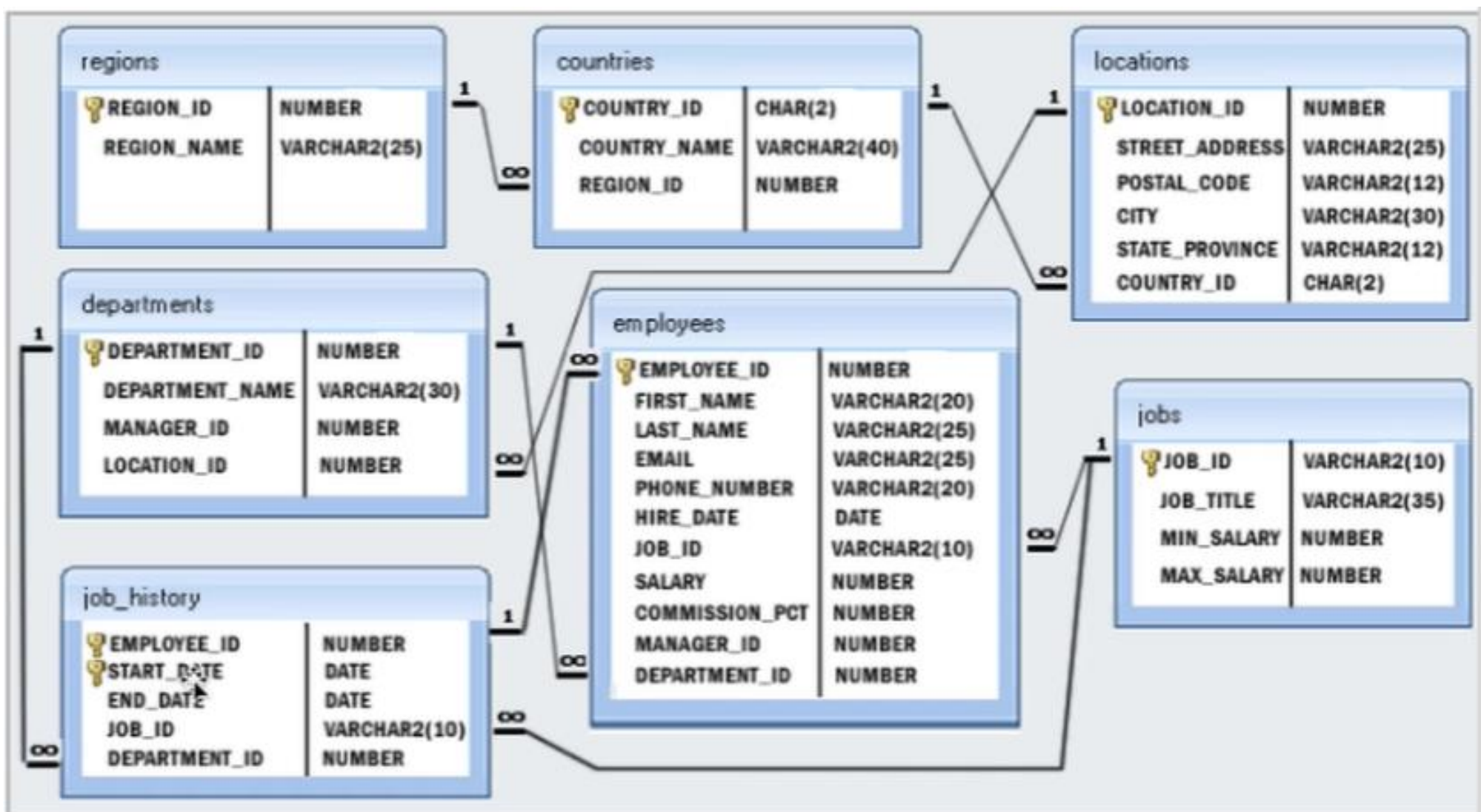
[38.21 Get Monthly Salary of each Employee](#)

[38.22 Calculate Average Salary with Total number of Employees](#)

38.23 Find Employees with 6 letter names

38.1 Human Resources Database Structure

Table	Function	Sample Data
regions	List of global regions	Americas, Europe
countries	List of countries	New York, Singapore
locations	List of sample locations address	123 valley road, New York
departments	List of departments in a company	Sales, Marketing
jobs	List of jobs	Admin, Sales, Marketing
Job_history	Job history information of an employee	Start Date, End Date, Job Id
employees	List of employees information	First Name, Last Name, Email



38.2 Quick Overview of All Tables

```
SELECT * FROM countries;
```

Showing rows: 1 to 25 Page No: 1 of 1

	country_id [PK] character varying (2)	country_name character varying (40)	region_id numeric (10)
1	AR	Argentina	2
2	AU	Australia	3
3	BE	Belgium	1
4	BR	Brazil	2

```
SELECT * FROM regions;
```

	region_id [PK] numeric (10)	region_name character (25)
1	1	Europe
2	2	Americas
3	3	Asia
4	4	Middle East and Africa

FK

```
SELECT * FROM locations;
```

SQL Showing rows: 1 to 15 Page No: 1 of 1

	location_id [PK] numeric (4)	street_address character varying (40)	postal_code character varying (12)	city character varying (30)	state_province character varying (25)	country_id character varying (2)
1	1000	1297 Via Cola di Rie	989	Roma	[default]	IT
2	1100	93001 Calle della Testa	10934	Venice	[default]	IT
3	1200	2017 Shinjuku-ku	1689	Tokyo	Tokyo Prefecture	JP
4	1300	9450 Kamiyacho	6823	Hiroshima	[default]	JP
5	1400	2014 Jabbarwadi Rd	26102	Southlake	Texas	US

FK

Showing rows: 1 to 27 Page No: 1 of 1

	department_id [PK] numeric (4)	department_name character varying (30)	manager_id numeric (6)	location_id numeric (4)
1	10	Administration	200	1700
2	20	Marketing	201	1800
3	30	Purchasing	114	1700
4	40	Human Resources	203	2400
5	50	Shipping	121	1500

```
SELECT * FROM departments;
```

```
SELECT * FROM jobs;
```

Showing rows: 1 to 19 Page No: 1 of 1

	job_id [PK] character varying (10)	job_title character varying (35)	min_salary numeric (6)	max_salary numeric (6)
1	AD_PRES	President	20000	40000
2	AD_VP	Administration Vice President	15000	30000
3	AD_ASST	Administration Assistant	3000	6000
4	FI_MGR	Finance Manager	8200	16000
5	FI_ACCOUNT	Accountant	4200	9000
6	AC_MGR	Accounting Manager	8200	16000
7	AC_ACCOUNT	Public Accountant	4200	9000

FK

```
SELECT * FROM job_history;
```

Showing rows: 1 to 10 Page No: 1 of 1

	employee_id numeric (6)	start_date date	end_date date	job_id character varying (10)	department_id numeric (4)
1	102	1993-01-13	2010-07-24	IT_PROG	60
2	101	1989-09-21	2010-10-27	AC_ACCOUNT	110
3	101	1993-10-28	2020-03-15	AC_MGR	
4	201	1996-02-17	1999-12-19	MK_REP	
5	114	1998-02-24	1998-12-31	ST_CLERK	

FK

```
SELECT * FROM departments;
```

Showing rows: 1 to 27 Page No: 1 of 1

	department_id [PK] numeric (4)	department_name character varying (30)	manager_id numeric (6)	location_id numeric (4)
1	10	Administration	200	1700
2	20	Marketing	201	1800
3	30	Purchasing	114	1700
4	40	Human Resources	203	2400
5	50	Shipping	121	1500

```
SELECT * FROM jobs;
```

Showing rows: 1 to 19  Page No: 1 of 1    

	job_id [PK] character varying (10) 	job_title character varying (35) 	min_salary numeric (6) 	max_salary numeric (6) 
1	AD_PRES	President	20000	
2	AD_VP	Administration Vice President	15000	
3	AD_ASST	Administration Assistant	3000	
4	FI_MGR	Finance Manager	8200	
5	FI_ACCOUNT	Accountant	4200	
6	AC_MGR	Accounting Manager	8200	
7	AC_ACCOUNT	Public Accountant	4200	

```
SELECT * FROM departments;
```

Showing rows: 1 to 27  Page No: 1 of 1    

	department_id [PK] numeric (4) 	department_name character varying (30) 	manager_id numeric (6) 	location_id numeric (4) 
1	10	Administration	200	1700
2	20	Marketing	201	1800
3	30	Purchasing	114	1700
4	40	Human Resources	203	2400
5	50	Shipping	121	1500

FK

FK

```
SELECT * FROM employees;
```

	employee_id [PK] numeric (6) 	first_name character varying (20) 	last_name character varying (25) 	email character varying (125) 	phone_number character varying (20) 	hire_date date 	job_id character varying (10) 	salary numeric (8,2) 	commission_pct numeric (2,2) 	manager_id numeric (6)	department_id numeric (4)
1	100	Steven	King	steven.king@company.com	515.123.4567	2010-06-17	AD_PRES	24000.00	0.00	0	90
2	101	Neena	Kochhar	neena.kochhar@company.com	515.123.4568	2010-06-18	AD_VP	17000.00	0.00	100	90
3	102	Lex	De Haan	lex.de.haan@company.com	515.123.4569	2010-06-19	AD_VP	17000.00	0.00	100	90
4	103	Alexander	Hunold	alexander.hunold@company.com	590.423.4567	2010-06-20	IT_PROG	9000.00	0.00	102	60
5	104	Bruce	Ernst	bruce.ernst@company.com	590.423.4568	2010-06-21	IT_PROG	6000.00	0.00	103	60
6	105	David	Austin	david.austin@company.com	590.423.4569	2011-06-22	IT_PROG	4800.00	0.00	103	60
7	106	Valli	Pataballa	valli.pataballa@company.com	590.423.4560	2011-06-23	IT_PROG	4800.00	0.00	103	60
8	107	Diana	Lorentz	diana.lorentz@company.com	590.423.5567	2011-06-24	IT_PROG	4200.00	0.00	103	60
9	114	Den	Raphaely	den.raphaely@company.com	515.127.4561	2011-07-01	FI_MGR	11000.00	0.00	100	90

38.3 Get Highest, Lowest, Total & average salaries of Employees

	min_salary numeric	max_salary numeric	total_salary numeric	avg_salary numeric
1	2100.00	24000.00	691400.00	6461.68

```
SELECT * FROM SELECT
      ROUND(MIN(salary),2) as min_salary,
      ROUND(MAX(salary),2) as max_salary,
      SUM(salary) as total_salary,
      ROUND(AVG(salary),2) as avg_salary
FROM employees
employees;
```

38.4 Difference between Highest & Lowest salaries by job_id

```
SELECT
  job_id,
  ROUND(MIN(salary),2) as min_salary,
  ROUND(MAX(salary),2) as max_salary,
  ROUND((MAX(salary) - MIN(salary)),2) as salary_diff
FROM employees
GROUP BY job_id;
```

Showing rows: 1 to 19		Page No: 1	of 1			
	job_id character varying (10)	min_salary numeric	max_salary numeric	salary_diff numeric		
1	SH_CLERK	2500.00	4200.00	1700.00		
2	AD_VP	17000.00	17000.00	0.00		
3	SA_MAN	10500.00	14000.00	3500.00		
4	PU_MAN	11000.00	11000.00	0.00		
5	IT_PROG	4200.00	9000.00	4800.00		
6	ST_CLERK	2100.00	3600.00	1500.00		

38.5 Get lowest paid salaries by each Manager

```
SELECT
    manager_id,
    salary,
    MIN(salary)
FROM employees
WHERE manager_id IS NOT NULL
AND manager_id != 0
GROUP BY manager_id, salary
ORDER BY MIN(salary) DESC;
```

Showing rows: 1 to 96  Page No: 1

	manager_id numeric (6)	salary numeric (8,2)	min numeric
1	100	17000.00	17000.00
2	100	14000.00	14000.00
3	100	13500.00	13500.00
4	100	13000.00	13000.00
5	101	12000.00	12000.00

38.6 Average Salary for each Department with more than 10 employees

```
SELECT
    department_id,
    ROUND(AVG(salary),0) as avg_salary,
    COUNT(*) as total_employees
FROM employees
GROUP BY department_id
HAVING COUNT(*) > 10;
```

	department_id numeric (4)	avg_salary numeric	total_employees bigint
1	80	8956	34
2	50	3476	45

38.9 Average Salary for each Department excluding Programmers

```
SELECT
    job_id,
    ROUND(AVG(salary),2) as avg_salary
FROM employees
WHERE job_id <> ('IT_PROG')
GROUP BY job_id
ORDER BY 2 DESC;
```

Showing rows: 1 to 18 			Page No: 1	
	job_id character varying (10) 	avg_salary numeric 		
1	AD_PRES	24000.00		
2	AD_VP	17000.00		
3	MK_MAN	13000.00		
4	SA_MAN	12200.00		
5	FI_MGR	12000.00		

38.10 Max Salary for each post where salary is equal or above \$5000

```
SELECT
    job_id,
    MAX(salary) as max_salary
FROM employees
GROUP BY job_id
HAVING MAX(salary) >= 5000
ORDER BY 2 DESC;
```

Showing rows: 1 to 15 			Page No: 1	
	job_id character varying (10) 	max_salary numeric 		
1	AD_PRES	24000.00		
2	AD_VP	17000.00		
3	SA_MAN	14000.00		
4	MK_MAN	13000.00		
5	AC_MGR	12000.00		
6	FI_MGR	12000.00		

38.9 Using Alias name with Columns

```
SELECT
  first_name as "First Name",
  last_name as "Last Name"
FROM employees;
```

Showing rows: 1 to 107 			Page No: 1	
	First Name character varying (20) 	Last Name character varying (25) 		
1	Steven	King		
2	Neena	Kochhar		
3	Lex	De Haan		
4	Alexander	Hunold		
5	Bruce	Ernst		

38.10 Compute 15% of salaries for all employees

```
SELECT
  first_name as "First Name",
  last_name as "Last Name",
  salary,
  salary * 0.15 as "salary_15%"
FROM employees
ORDER BY 3 DESC;
```

Showing rows: 1 to 107 			Page No: 1		of 1					
	First Name character varying (20) 	Last Name character varying (25) 	salary numeric (8,2) 	salary_15% numeric 						
1	Steven	King	24000.00	3600.0000						
2	Neena	Kochhar	17000.00	2550.0000						
3	Lex	De Haan	17000.00	2550.0000						
4	John	Russell	14000.00	2100.0000						
5	Karen	Partners	13500.00	2025.0000						

38.11 List all employee ids with each job id group

```
SELECT
  job_id,
  ARRAY_AGG(employee_id) as "All Employees IDs"
FROM employees
GROUP BY job_id;
```

	job_id character varying (10)	All Employees IDs numeric[]
1	SH_CLERK	{180,181,182,183,184,185,186,187,188,189,190,191,192,193,194,195,196,197,198,199}
2	AD_VP	{101,102}
3	SA_MAN	{145,146,147,148,149}
4	PU_MAN	{114}
5	IT_PROG	{103,104,105,106,107}
6	ST_CLERK	{122,124,125,126,127,128,129,130,140,120,120,121,122,141,142,143,144,125,126,127,128}

38.12 Discard Characters from Employee Email Address

```
SELECT
  first_name,
  email,
  length(email),
  substr(email,1,length(email)-4) as "extracted email"
FROM employees
```

	first_name character varying (20)	email character varying (125)	length integer	extracted email text
1	Steven	steven.king@company.com	23	steven.king@company
2	Neena	neena.kochhar@company.com	25	neena.kochhar@company
3	Lex	lex.de haan@company.com	23	lex.de haan@company
4	Alexander	alexander.hunold@company.com	28	alexander.hunold@company

38.13 List all Employees with First name beginning with A, B or M

```
SELECT
    first_name,
    last_name
FROM employees
WHERE first_name LIKE 'A%'
OR first_name LIKE 'B%'
OR first_name LIKE 'M%';
```

Showing rows: 1 to 18 			Page No: 1	
	first_name character varying (20) 	last_name character varying (25) 		
1	Alexander	Hunold		
2	Bruce	Ernst		
3	Alexander	Khoo		
4	Michael	Rogers		
5	Mozhe	Atkinson		

38.14 String Manipulation with Upper, Lower & initcap functions

```
SELECT
    UPPER(first_name),
    LOWER(first_name),
    INITCAP(first_name)
FROM employees;
```

Showing rows: 1 to 107 				Page No: 1	
	upper text 	lower text 	initcap text 		
1	STEVEN	steven	Steven		
2	NEENA	neena	Neena		
3	LEX	lex	Lex		
4	ALEXANDER	alexander	Alexander		
5	BRUCE	bruce	Bruce		
6	DAVID	david	David		

38.15 Using Substring to get Portion of string data

```
SELECT
  SUBSTRING(first_name, 1, 3)
FROM employees;
```

Showing rows: 1 to 107 

	substring text
1	Ste
2	Nee
3	Lex
4	Ala

38.16 Get Unique Designations in Employees table

```
SELECT
  count (DISTINCT job_id) as "Unuique designations"
FROM employees;
```

	Unuique designations bigint
1	19

38.17 Get Employees with particular department id

```
SELECT
  first_name as "First Name",
  last_name as "Last Name",
  department_id as "department ID"
FROM employees
WHERE department_id IN (30,100)
ORDER BY department_id;
```

Showing rows: 1 to 12  Page No: 1 of 1  

	First Name character varying (20)	Last Name character varying (25)	department ID numeric (4)
1	Den	Raphaely	30
2	Alexander	Khoo	30
3	Shelli	Baida	30
4	Sigal	Tobias	30
5	Guy	Himuro	30
6	Karen	Colmenares	30

38.18 Select Employees not in Range

```
SELECT
    first_name,
    last_name,
    salary,
    department_id
FROM employees
WHERE department_id NOT IN
(30,100)
AND salary NOT BETWEEN
(10000) AND (20000);
```

Showing rows: 1 to 79  Page No: 1 of 1    

	first_name character varying (20) 	last_name character varying (25) 	salary numeric (8,2) 	department_id numeric (4) 
1	Steven	King	24000.00	90
2	Alexander	Hunold	9000.00	60
3	Bruce	Ernst	6000.00	60
4	David	Austin	4800.00	60
5	Valli	Pataballa	4800.00	60

38.19 Find 'c' in last name at 3rd or greater position

```
SELECT
    first_name,
    last_name,
    POSITION('C' IN last_name )
FROM employees
WHERE POSITION('C' IN last_name ) > 2;
```

Showing rows: 1 to 1  Page No: 1 of 1  

	first_name character varying (20) 	last_name character varying (25) 	position integer 
1	Samuel	McCain	3

38.20 Update Phone Number with strings

```
UPDATE employees
SET phone_number = REPLACE(phone_number,
'123', '888')
WHERE phone_number LIKE '%123%';
UPDATE 16
Query returned successfully in 85 msec.
```

38.21 Get Monthly Salary of each Employee

```
SELECT
    first_name,
    last_name,
    salary as "Anual Salary",
    ROUND(salary/12,2) as "Monthly Salary"
FROM employees;
```

Showing rows: 1 to 107  Page No: 1 of 1   

	first_name character varying (20)	last_name character varying (25)	Anual Salary numeric (8,2)	Monthly Salary numeric
1	Alexander	Hunold	9000.00	750.00
2	Bruce	Ernst	6000.00	500.00

38.22 Calculate Average Salary with Total number of Employees

```
SELECT
    ROUND(AVG(salary),2) as avg_salary,
    COUNT(*)
FROM employees;
```

	avg_salary numeric 	count bigint 
1	6461.68	107

38.23 Find Employees with 6 letter names

```
SELECT
    first_name,
    last_name
FROM employees
WHERE length(last_name) = 6;
```

```
SELECT
    first_name,
    last_name
FROM employees
WHERE last_name LIKE '_____';
```

Showing rows: 1 to 28  Page No: 1 of 1

	first_name character varying (20) 	last_name character varying (25) 
1	Alexander	Hunold
2	David	Austin
3	Sigal	Tobias
4	Daniel	Faviet
5	Jason	Mallin
6	Michael	Baer

39. Exploring Stock Markets Data

39.1 Select first or last records in a table

39.2 Get the first or last record for each group

39.3 How to calculate cube root in PostgreSQL

39.1 Select first or last records in a table

```
SELECT
  *
FROM stocks_prices
WHERE symbol_id = 1
ORDER BY price_date
LIMIT 1;
```

	price_id [PK] integer	symbol_id integer	price_date date	open_price numeric (30,4)	high_price numeric (30,4)	low_price numeric (30,4)	close_price numeric (30,4)	change_1d numeric (30,4)	return_1d numeric (30,4)
1	1	1	2010-01-04	4.5621	4.6564	4.3925	4.4962	-0.0660	-1.4463

```
SELECT
  *
FROM stocks_prices
WHERE symbol_id = 1
ORDER BY price_date
OFFSET 10
LIMIT 1;
```

	price_id [PK] integer	symbol_id integer	price_date date	open_price numeric (30,4)	high_price numeric (30,4)	low_price numeric (30,4)	close_price numeric (30,4)	change_1d numeric (30,4)	return_1d numeric (30,4)
1	11	1	2010-01-19	5.1843	5.4011	5.1843	5.3162	0.1320	2.5455

39.2 Get the first or last record for each group

```
SELECT
    symbol_id,
    MIN(price_date)
FROM stocks_prices
GROUP BY symbol_id
ORDER BY symbol_id;
```

	symbol_id integer	min date
1	1	2010-01-04
2	2	2010-01-04
3	3	2010-01-04
4	4	2010-01-04
5	5	2010-01-04
6	8	2014-03-27
7	10	2010-01-04
8	11	2010-01-04
9	12	2010-01-04
10	13	2010-06-29
11	14	2010-01-04
12	15	2010-01-04

```
SELECT
    symbol_id,
    MAX(price_date)
FROM stocks_prices
GROUP BY symbol_id
ORDER BY symbol_id;
```

	symbol_id integer	max date
1	1	2020-08-28
2	2	2020-08-28
3	3	2020-08-28
4	4	2020-08-28
5	5	2020-08-28
6	8	2020-08-28
7	10	2020-08-28
8	11	2020-08-28
9	12	2020-08-28
10	13	2020-08-28
11	14	2020-08-28
12	15	2020-08-28

39.3 How to calculate cube root in PostgreSQL

```
SELECT  
    CBRT(close_price)  
FROM stocks_prices  
WHERE symbol_id = 1  
ORDER BY price_date DESC;
```

Showing rows: 1 to 1000 

	cbrt double precision 
1	2.386381199623365
2	2.3680962950857967
3	2.3383000778587806
4	2.3597452564256542
5	2.3775687811052517
6	2.299558831577932
7	2.3207944168063896
8	2.3318811826088286
9	2.3269667714505617
10	2.328197319946153
11	2.3710645813404065
12	2.360004500156760

40. Managing PostgreSQL Security

[40.1 Security Concepts & levels](#)

[40.2 Instance Level Security](#)

[40.3 Instance Level Security – Add Users to Roles](#)

[40.4 Use pgAdmin to Create Roles](#)

[40.5 Database Level Security](#)

[40.6 Schema Level Security](#)

[40.7 Table Level Security](#)

[40.8 Column level Security](#)

[40.9 Row Level Security](#)

[40.10 Using CURRENT USER with RLS](#)

[40.11 Row Level Policy for application users](#)

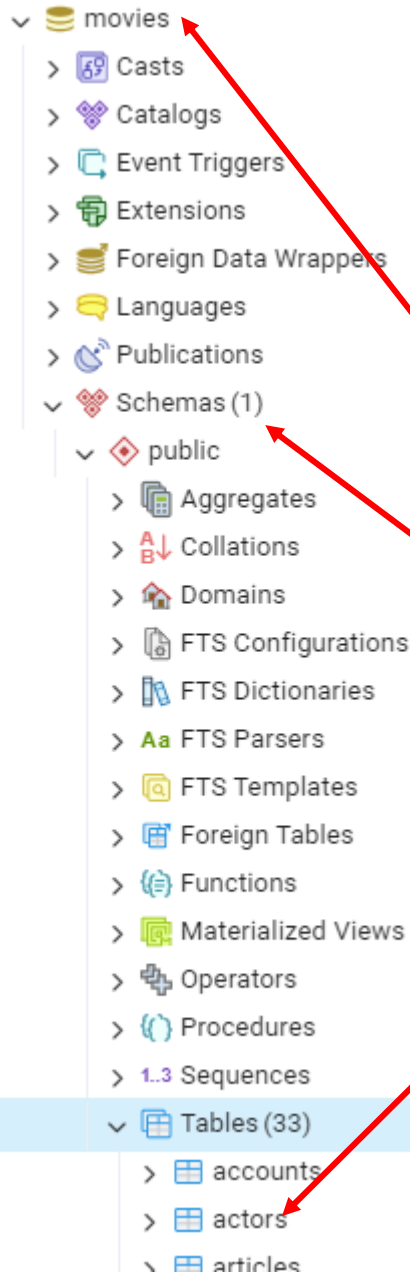
[40.12 Drop a Policy](#)

[40.13 Inspecting Permissions](#)

[40.14 Row Level Security Performance](#)

[40.15 Encrypted Data in Columns](#)

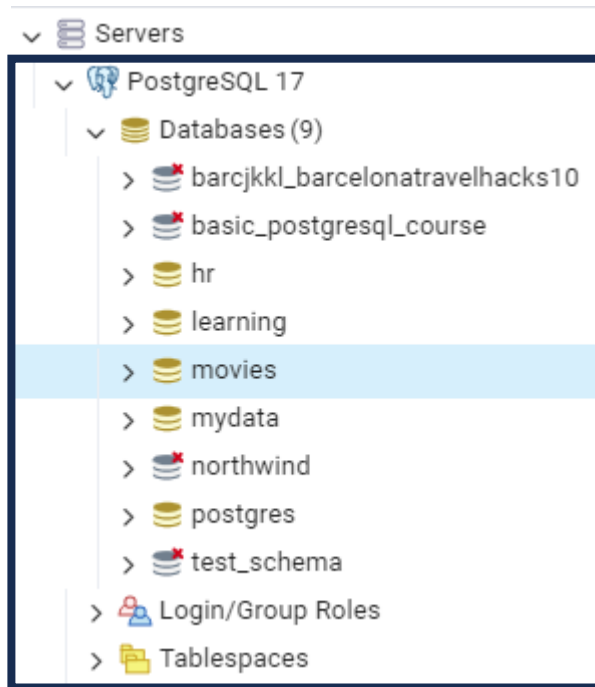
40.1 Security Concepts & levels



1. Security is set by roles.
2. A database System Administrator can grant or revoke access
3. Roles are everything!
4. Privileges are granted to users by means of 'Special Roles'
5. Security guards have different 'roles' as an information clerks.
6. There are 6 levels of security in PostgreSQL

1. Instance Level
2. Database Level
3. Schema Level
4. Table Level
5. Column level
6. Row level

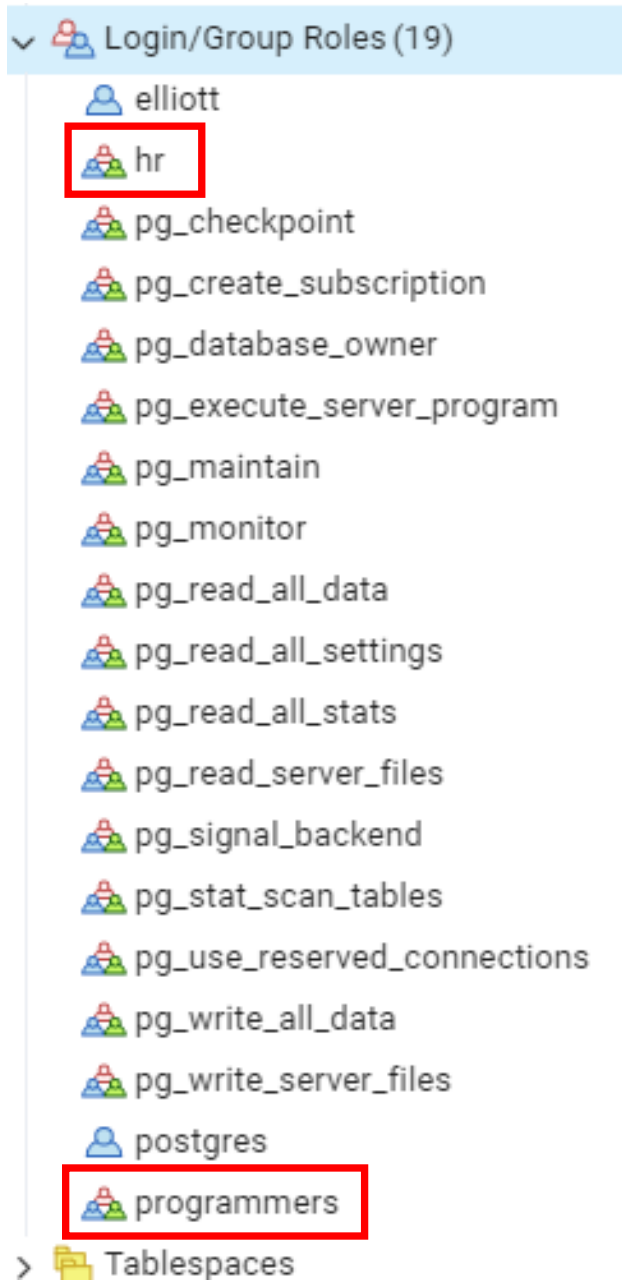
40.2 Instance Level Security



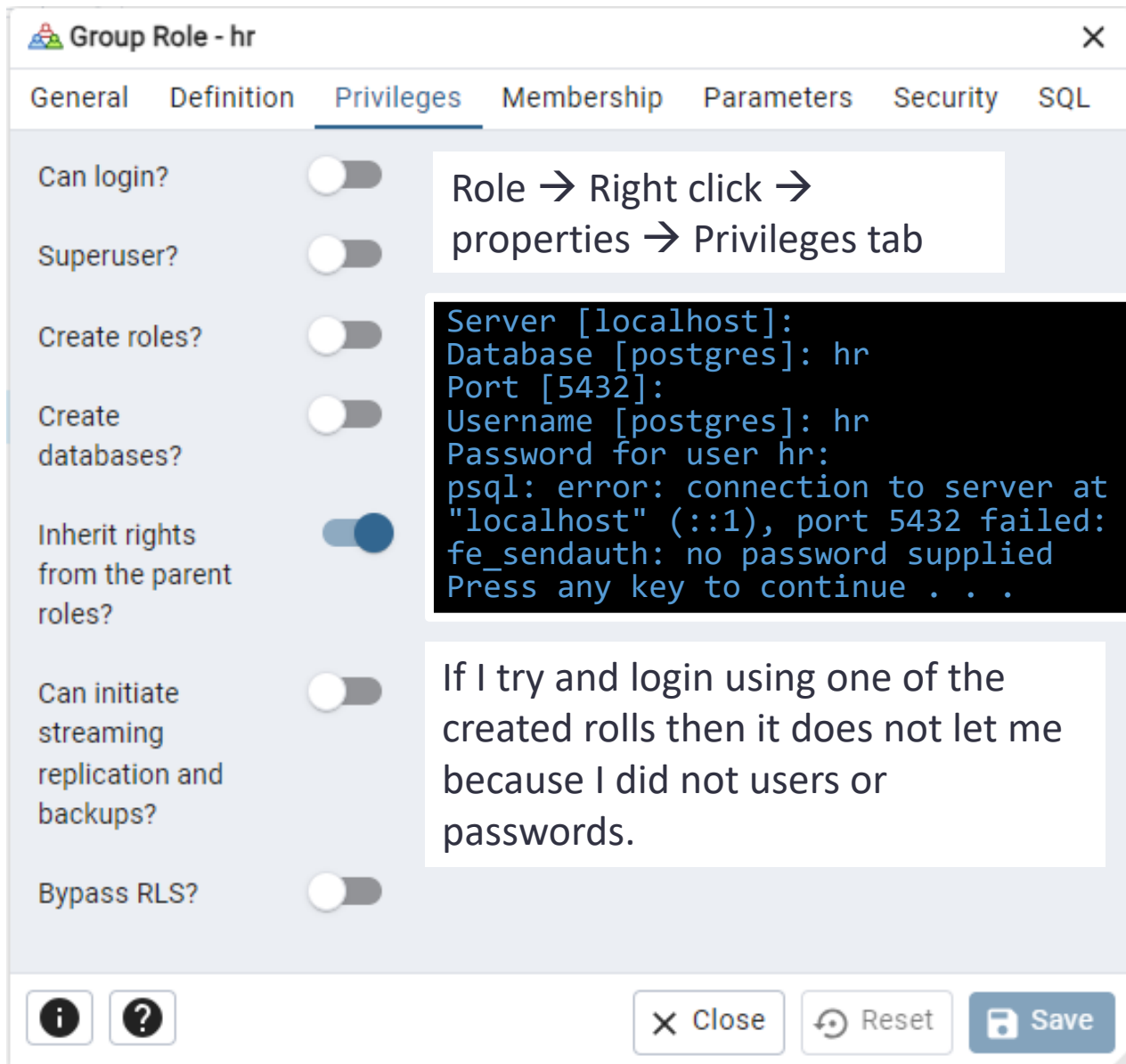
1. Controls all databases on an instance
2. Includes everything below server level
3. Highest level of security so be careful!
4. Can be assigned with:
 - SUPERUSER - super access, all access
 - CREATE DB - can create databases
 - CREATEROLE - Can create roles
 - LOGIN - Can login to a database
 - REPLICATION - Can be used for replication
4. Using NO means no access i.e. NOSUPERUSER = no superuser access

Create Role using `CREATE ROLE role_name privileges_list`. I will create the following roles in the HR database.

```
CREATE ROLE hr NOSUPERUSER NOCREATEDB NOCREATEROLE NOLOGIN;  
CREATE ROLE programmers NOSUPERUSER NOCREATEDB NOCREATEROLE NOLOGIN;
```



In PgAdmin4, if I go to the bottom, under login/Group roles I can see the two roles I just created. The Pg_ roles are system roles.



40.3 Instance Level Security – Add Users to Roles

Can I login using a role?

```
CREATE ROLE john NOSUPERUSER NOCREATEDB NOCREATEROLE LOGIN PASSWORD 'password123';  
CREATE ROLE linda NOSUPERUSER NOCREATEDB NOCREATEROLE LOGIN PASSWORD 'password123';
```

I have created the above users and can login to the HR database.

The screenshot displays a database management interface. On the left, a tree view shows the database structure. Under 'Servers (1)', 'PostgreSQL 17' is expanded, showing 'Databases (9)' and 'Login/Group Roles (21)'. The 'john' role is highlighted in the 'Login/Group Roles' list. On the right, the 'Login Role - john' configuration window is open, showing the 'Privileges' tab. The 'Can login?' toggle is turned on and highlighted with a red box. Other toggles include 'Superuser?', 'Create roles?', 'Create databases?', 'Inherit rights from the parent roles?', 'Can initiate streaming replication and backups?', and 'Bypass RLS?'. The 'Can login?' toggle is currently turned on.

Privilege	Status
Can login?	On
Superuser?	Off
Create roles?	Off
Create databases?	Off
Inherit rights from the parent roles?	On
Can initiate streaming replication and backups?	Off
Bypass RLS?	Off

```
Server [localhost]:  
Database [postgres]: hr  
Port [5432]:  
Username [postgres]: john  
Password for user john:  
psql (17.2)  
WARNING: Console code page (850) differs from Windows code page (1252)  
         8-bit characters might not work correctly. See psql reference  
         page "Notes for Windows users" for details.  
Type "help" for help.  
  
hr=>
```

However, I did not assign any privileges for 'john' for our database. It works because the database is under the public schema, which by default allows everyone to be able to login to our database so I should revoke all privileges on the public schema.

```
REVOKE ALL ON DATABASE hr FROM public;
```

```
Server [localhost]:  
Database [postgres]: hr  
Port [5432]:  
Username [postgres]: john  
Password for user john:  
psql: error: connection to server at "localhost" (:::1), port 5432  
failed: FATAL:  permission denied for database "hr"  
DETAIL:  User does not have CONNECT privilege.  
Press any key to continue . . .
```

Permission denied to login to HR from the public schema.

I need to grant the HR role to John and Linda.

```
GRANT hr TO john;  
GRANT hr TO Linda;
```

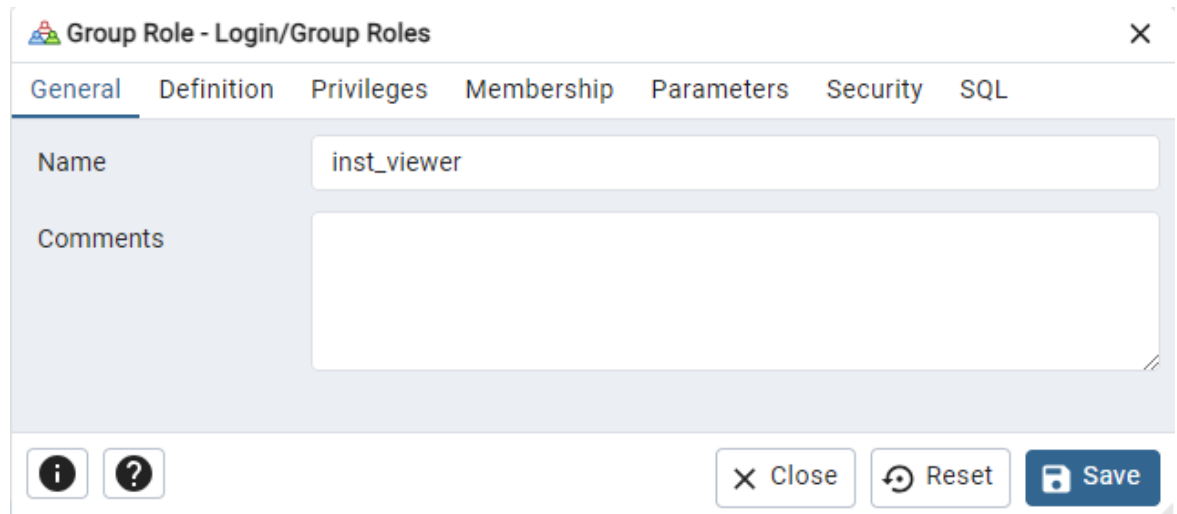
```
Server [localhost]:  
Database [postgres]: hr  
Port [5432]:  
Username [postgres]: john  
Password for user john:  
psql: error: connection to server at "localhost" (:::1), port 5432  
failed: FATAL:  permission denied for database "hr"  
DETAIL:  User does not have CONNECT privilege.  
Press any key to continue . . .
```

However, the role of hr is missing connect privilege.

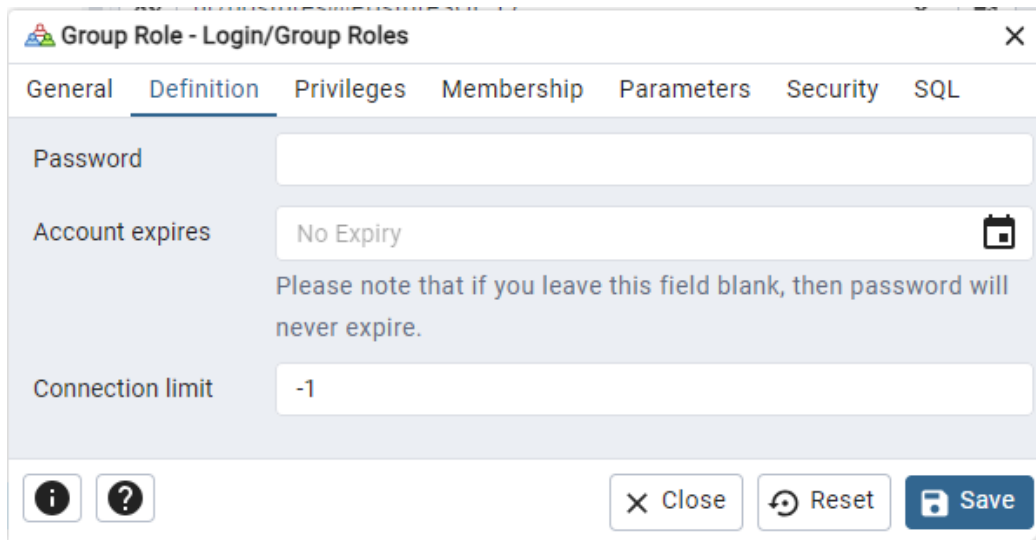
40.4 Use pgAdmin to Create Roles

From the instance level →
Login/Group roles → Right click
→ create role.

Give role a name i.e. inst_viewer



The screenshot shows the 'Group Role - Login/Group Roles' dialog box in pgAdmin, with the 'General' tab selected. The 'Name' field contains 'inst_viewer'. The 'Comments' field is empty. At the bottom, there are buttons for 'Close', 'Reset', and 'Save'.



The screenshot shows the 'Group Role - Login/Group Roles' dialog box in pgAdmin, with the 'Definition' tab selected. The 'Password' field is empty. The 'Account expires' field is set to 'No Expiry' with a calendar icon. Below it, a note states: 'Please note that if you leave this field blank, then password will never expire.' The 'Connection limit' field is set to '-1'. At the bottom, there are buttons for 'Close', 'Reset', and 'Save'.

I don't want this role to have a password so I leave it blank

Group Role - Login/Group Roles

General Definition Privileges Membership Parameters Security SQL

Can login? ☒

Superuser? ☐

Create roles? ☐

Create databases? ☐

Inherit rights from the parent roles? ☒

Can initiate streaming replication and backups? ☐

(Information and Help icons)

In the privileges tab I can set the rights for this role.

In the SQL tab it shows the SQL that is used to create this role.

Group Role - Login/Group Roles

General Definition Privileges Membership Parameters Security SQL

```
1 CREATE ROLE inst_viewer WITH
2   LOGIN
3   NOSUPERUSER
4   NOCREATEDB
5   NOCREATEROLE
6   INHERIT
7   NOREPLICATION
8   NOBYPASSRLS
9   CONNECTION LIMIT -1;
```

(Information and Help icons) Close Reset Save

```
CREATE ROLE inst_editor WITH  
  LOGIN  
  NOSUPERUSER  
  CREATEDB  
  CREATEROLE  
  INHERIT  
  NOREPLICATION  
  NOBYPASSRLS  
  CONNECTION LIMIT -1;
```

```
CREATE ROLE inst_superuser WITH  
  LOGIN  
  SUPERUSER  
  CREATEDB  
  CREATEROLE  
  INHERIT  
  REPLICATION  
  BYPASSRLS  
  CONNECTION LIMIT -1;
```

I create two more roles, one is instance level editor and the other is instance level Superuser.

40.5 Database Level Security

The database level security is controlled at a database level and has three levels:

CREATE – make a new schema

CONNECT – Connect to a database

TEMP/TEMPORARY – Create a temporary table

We can add a connection permission via CONNECT

```
GRANT CONNECT ON DATABASE hr TO hr;  
GRANT CONNECT ON DATABASE hr TO programmers;
```

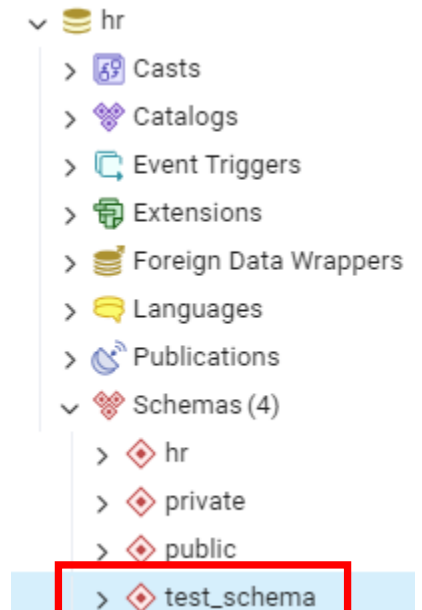
Now that I have granted the role of hr the connect privilege, all users in that role can connect to the database but they cannot create schemas because that privilege has not been granted.

```
Server [localhost]:  
Database [postgres]: hr  
Port [5432]:  
Username [postgres]: john  
Password for user john:  
psql (17.2)  
WARNING: Console code page (850) differs from Windows code page (1252)  
         8-bit characters might not work correctly. See psql reference  
         page "Notes for Windows users" for details.  
Type "help" for help.  
  
hr=>  
hr=> CREATE SCHEMA test_schema;  
ERROR:  permission denied for database hr  
hr=>
```

```
GRANT CREATE ON DATABASE hr TO hr;  
GRANT CREATE ON DATABASE hr TO programmers;
```

If I grant create privileges to the role then log out and log back in again, the users under that role now can create schemas.

```
Server [localhost]:  
Database [postgres]: hr  
Port [5432]:  
Username [postgres]: john  
Password for user john:  
psql (17.2)  
WARNING: Console code page (850) differs from Windows code page (1252)  
         8-bit characters might not work correctly. See psql reference  
         page "Notes for Windows users" for details.  
Type "help" for help.  
  
hr=> CREATE SCHEMA test_schema;  
CREATE SCHEMA  
hr->
```



40.6 Schema Level Security

The schema level security comes after the database in the tree and applies to the schema. There are two security privileges that we can grant to our roles at this level:

CREATE – Create tables, functions etc

USAGE – View schema and its objects

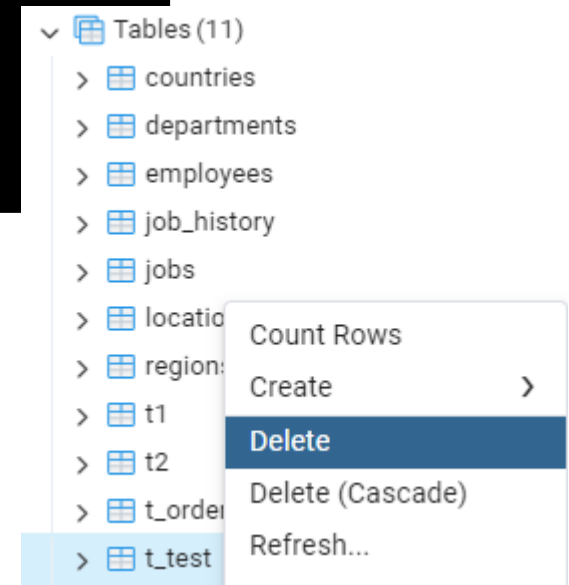
By default, all users have CREATE level access.

```
Server [localhost]:
Database [postgres]: hr
Port [5432]:
Username [postgres]: john
Password for user john:
psql (17.2)
WARNING: Console code page (850) differs from Windows code page (1252)
         8-bit characters might not work correctly. See psql reference
         page "Notes for Windows users" for details.
Type "help" for help.
hr=>
hr=> CREATE TABLE t_test (id INT);
CREATE TABLE
hr=>
```

```
REVOKE ALL ON SCHEMA public from public;
REVOKE
```

Query returned successfully in 37 msec.

After revoking all permissions on the public schema, I also delete the t_test table.



```

Server [localhost]:
Database [postgres]: hr
Port [5432]:
Username [postgres]: john
Password for user john:
psql (17.2)
WARNING: Console code page (850) differs from Windows code page (1252)
         8-bit characters might not work correctly. See psql reference
         page "Notes for Windows users" for details.
Type "help" for help.

hr=> CREATE TABLE t_test (id INT);
ERROR: no schema has been selected to create in
LINE 1: CREATE TABLE t_test (id INT);
                        ^

hr=> CREATE TABLE public.t_test (id INT);
ERROR: permission denied for schema public
LINE 1: CREATE TABLE public.t_test (id INT);
                        ^

hr=>

```

Now John who is part of the role hr cannot create tables in the public schema.

```

CREATE ROLE sales NOSUPERUSER NOCREATEDB NOCREATEROLE LOGIN PASSWORD 'welcome';
CREATE ROLE tech NOSUPERUSER NOCREATEDB NOCREATEROLE LOGIN PASSWORD 'welcome';

```

I have created two additional roles and granted usage permissions to the role of sales and create permissions to the role of tech.

```

GRANT USAGE ON SCHEMA public TO sales;
GRANT USAGE ON SCHEMA public TO tech;
GRANT CREATE ON SCHEMA public TO tech;

```



```
Server [localhost]:
Database [postgres]: hr
Port [5432]:
Username [postgres]: sales
Password for user sales:
psql: error: connection to server at "localhost" (:::1), port 5432
failed: FATAL: permission denied for database "hr"
DETAIL: User does not have CONNECT privilege.
Press any key to continue . . .
```

But when I try
and log into the
database with
these roles, I get
permission
denied. Why?

```
GRANT CONNECT ON DATABASE hr TO sales;
GRANT CONNECT ON DATABASE hr TO tech;
```

I have to grant database connection permission to
these roles to permit database connection.

```
Server [localhost]:
Database [postgres]: hr
Port [5432]:
Username [postgres]: sales
Password for user sales:
psql (17.2)
WARNING: Console code page (850) differs from Windows code page (1252)
        8-bit characters might not work correctly. See psql reference
        page "Notes for Windows users" for details.
Type "help" for help.

hr=> CREATE TABLE t_test (id INT);
ERROR: permission denied for schema public
LINE 1: CREATE TABLE t_test (id INT);
                        ^
hr=>
```

Sales role
cannot create a
table because
the role has
USAGE
permissions
only.

```
Server [localhost]:
Database [postgres]: hr
Port [5432]:
Username [postgres]: tech
Password for user tech:
psql (17.2)
WARNING: Console code page (850) differs from Windows code page (1252)
        8-bit characters might not work correctly. See psql reference
        page "Notes for Windows users" for details.
Type "help" for help.

hr=> CREATE TABLE t_test (id INT);
ERROR:  no schema has been selected to create in
LINE 1: CREATE TABLE t_test (id INT);
                        ^

hr=> CREATE TABLE public.t_test (id INT);
CREATE TABLE
hr=>
```

Tech role can
create a table in
the public
schema because
this role has
CREATE
permission

The key is to revoke all permissions at each level then go assigning the relevant permission for each role.

40.7 Table Level Security

Table level security has the following privileges that can be assigned to a role and user:

SELECT – read data/rows from table

INSERT – insert data into tables

UPDATE – update data into tables

DELETE – Delete data from tables

TRUNCATE – remove all rows/delete all rows at once (careful! This will empty a table)

TRIGGER – Create triggers on tables (high level access!!)

REFERENCE – Ability to create a foreign key constraint on a table

Granting permissions can be done to ALL tables with the following syntax:

GRANT permission_name ON ALL TABLES IN SCHEMA schema_name TO role_name

Granting permissions to an individual table can be done with the below syntax:

GRANT permission_name ON table_name TO role_name

```
GRANT SELECT ON ALL TABLES IN SCHEMA public TO sales;  
GRANT SELECT ON ALL TABLES IN SCHEMA public TO tech;
```

```
GRANT INSERT on TABLE employees TO tech;  
GRANT UPDATE on TABLE employees TO tech;  
GRANT DELETE on TABLE employees TO tech;
```

```
Server [localhost]:
Database [postgres]: hr
Port [5432]:
Username [postgres]: sales
Password for user sales:
psql (17.2)
```

The role of Sales has only been granted select permissions on all tables. So an insert query fails.

```
WARNING: Console code page (850) differs from Windows code page (1252)
        8-bit characters might not work correctly. See psql reference
        page "Notes for Windows users" for details.
```

```
Type "help" for help.
```

```
hr=> SELECT * FROM countries;
```

country_id	country_name	region_id
AR	Argentina	2
...		
ZW	Zimbabwe	4

(25 rows)

```
hr=> SELECT employee_id, first_name, last_name, email, phone_number FROM employees LIMIT 5;
```

employee_id	first_name	last_name	email	phone_number
103	Alexander	Hunold	alexander.hunold@company.com	590.423.4567
104	Bruce	Ernst	bruce.ernst@company.com	590.423.4568
105	David	Austin	david.austin@company.com	590.423.4569
106	Valli	Pataballa	valli.pataballa@company.com	590.423.4560
107	Diana	Lorentz	diana.lorentz@company.com	590.423.5567

(5 rows)

```
hr=> INSERT INTO regions (region_name) VALUES ('test');
```

```
ERROR: permission denied for table regions
```

```
Server [localhost]:  
Database [postgres]: hr  
Port [5432]:  
Username [postgres]: tech  
Password for user tech:  
psql (17.2)
```

The role of tech has been granted select, insert, update and delete permissions only on employees table.

```
WARNING: Console code page (850) differs from Windows code page (1252)  
8-bit characters might not work correctly. See psql reference  
page "Notes for Windows users" for details.
```

```
Type "help" for help.
```

```
hr=> SELECT * FROM regions;
```

region_id	region_name
1	Europe
2	Americas
3	Asia
4	Middle East and Africa

```
(4 rows)
```

```
hr=> INSERT INTO regions (region_name) VALUES ('test');
```

```
hr=> INSERT INTO regions (region_name) values ('test');
```

```
ERROR: permission denied for table regions
```

```
hr=> INSERT INTO public.regions (region_name) values ('test');
```

```
ERROR: permission denied for table regions
```

```
hr=>
```

Insert query for any table other than employees fails permissions.

```
GRANT INSERT on TABLE regions TO tech;
GRANT UPDATE on TABLE regions TO tech;
GRANT DELETE on TABLE regions TO tech;
```

```
Server [localhost]:
Database [postgres]: hr
Port [5432]:
Username [postgres]: tech
Password for user tech:
psql (17.2)
```

```
WARNING: Console code page (850) differs from Windows code page (1252)
         8-bit characters might not work correctly. See psql reference
         page "Notes for Windows users" for details.
Type "help" for help.
```

After granting INSERT permissions for table regions to tech, the tech role can now insert a row.

```
hr=> INSERT INTO regions (region_name) VALUES ('test');
ERROR:  null value in column "region_id" of relation "regions" violates not-null constraint
DETAIL:  Failing row contains (null, test).
```

```
hr=> INSERT INTO regions (region_id, region_name) VALUES (10, 'test');
INSERT 0 1
```

```
hr=>
```

```
hr=> DELETE FROM regions WHERE region_id = 10;
DELETE 1
```

```
hr=> SELECT * FROM regions;
```

region_id	region_name
1	Europe
2	Americas
3	Asia
4	Middle East and Africa

```
(4 rows)
```

Tech role also has delete permissions for table regions so I will do housekeeping by deleting the region of test.

40.7 Column Level Security

Column level security has the following privileges that can be assigned to a role and user:

SELECT – read data from a column

INSERT – insert data into a column

UPDATE – update data in a column

REFERENCE – Ability to refer to the column as a foreign key

There is no DELETE privilege from the column

Granting column level permissions can be done to a table with the following syntax:

GRANT permission_name (table_col1, table_col2, ...) ON table_name TO role_name

```
Server [localhost]:
```

```
Database [postgres]: hr
```

```
Port [5432]:
```

```
Username [postgres]: sales
```

```
Password for user sales:
```

```
psql (17.2)
```

```
WARNING: Console code page (850) differs from Windows code page (1252)
```

```
8-bit characters might not work correctly. See psql reference
```

```
page "Notes for Windows users" for details.
```

```
Type "help" for help.
```

```
hr=> SELECT employee_id, first_name, last_name, phone_number, salary FROM employees limit 4;
```

employee_id	first_name	last_name	phone_number	salary
103	Alexander	Hunold	590.423.4567	9000.00
104	Bruce	Ernst	590.423.4568	6000.00
105	David	Austin	590.423.4569	4800.00
106	Valli	Pataballa	590.423.4560	4800.00

```
(5 rows)
```

For example I want to restrict sales role from having access to employees→phone_number, salary etc.

```
REVOKE SELECT ON ALL TABLES IN SCHEMA public FROM sales;
```

```
Server [localhost]:
Database [postgres]: hr
Port [5432]:
Username [postgres]: sales
Password for user sales:
psql (17.2)
...
hr=> SELECT * FROM employees;
ERROR: permission denied for table employees
hr=>
```

```
GRANT SELECT (employee_id, first_name, last_name, email) ON employees TO sales;
```

```
Server [localhost]:
Database [postgres]: hr
Port [5432]:
Username [postgres]: sales
Password for user sales:
psql (17.2)
...
hr=> SELECT * FROM employees;
ERROR: permission denied for table employees
hr=>
hr=> SELECT employee_id, first_name, last_name, email FROM employees
limit 3;
 employee_id | first_name | last_name | email
-----+-----+-----+-----
          103 | Alexander | Hunold   | alexander.hunold@company.com
          104 | Bruce    | Ernst    | bruce.ernst@company.com
          105 | David    | Austin   | david.austin@company.com
(3 rows)
```

Because the sales role only has permission to view 4 columns in the employees table, selecting all columns fails with a permission denied error.

```
hr=>
```

However, selecting only the columns for which permission has been granted works


```
GRANT SELECT (region_id, region_name) ON regions TO sales;
```

```
Server [localhost]:
```

```
Database [postgres]: hr
```

```
Port [5432]:
```

```
Username [postgres]: sales
```

```
Password for user sales:
```

```
psql (17.2)
```

```
WARNING: Console code page (850) differs from Windows code page (1252)  
8-bit characters might not work correctly. See psql reference  
page "Notes for Windows users" for details.
```

```
Type "help" for help.
```

```
hr=> SELECT * FROM regions;
```

region_id	region_name
1	Europe
2	Americas
3	Asia
4	Middle East and Africa

```
(4 rows)
```

I grant SELECT only permission to sales for all columns of the table. UPDATE query fails permissions because I have not granted update permissions on the columns of this table.

```
hr=> UPDATE regions SET region_name = 'Asia1' WHERE region_id = 3;
```

```
ERROR: permission denied for table regions
```

```
hr=>
```

```
GRANT UPDATE (region_name) ON regions TO sales;
```

```
Server [localhost]:
Database [postgres]: hr
Port [5432]:
Username [postgres]: sales
Password for user sales:
psql (17.2)
WARNING: Console code page (850) differs from Windows code page (1252)
        8-bit characters might not work correctly. See psql reference
        page "Notes for Windows users" for details.
Type "help" for help.
```

```
hr=> UPDATE regions SET region_name = 'Asia1' WHERE region_id = 3;
UPDATE 1
```

```
hr=> SELECT * FROM regions;
 region_id |      region_name
-----+-----
          1 | Europe
          2 | Americas
          4 | Middle East and Africa
          3 | Asia1
(4 rows)
```

I grant UPDATE permission on a particular column of the regions table and now sales can modify the data within that column.

```
hr=> UPDATE regions SET region_name = 'Asia' WHERE region_id = 3;
UPDATE 1
```

```
hr=> SELECT * FROM regions;
 region_id |      region_name
-----+-----
          1 | Europe
          2 | Americas
          4 | Middle East and Africa
          3 | Asia
(4 rows)
```

40.8 Row Level Security

Row level security must be enabled at table level because by default RLS is disabled.

```
ALTER TABLE table_name ENABLE ROW LEVEL SECURITY
```

After RLS is enabled on a table the default policy value is Deny all.

```
GRANT SELECT ON TABLE jobs TO sales;
```

```
Server [localhost]:
Database [postgres]: hr
Port [5432]:
Username [postgres]: sales
Password for user sales:
psql (17.2)
WARNING: Console code page (850) differs from Windows code page (1252)
         8-bit characters might not work correctly. See psql reference
         page "Notes for Windows users" for details.
Type "help" for help.

hr=> SELECT * FROM jobs;
ERROR: permission denied for table jobs
hr=> SELECT * FROM jobs;
 job_id | job_title | min_salary | max_salary
-----+-----+-----+-----
 AD_PRES | President |      20000 |      40000
 AD_VP   | Administration Vice President |      15000 |      30000
 ...
 PR_REP  | Public Relations Representative |       4500 |      10500
(19 rows)

hr=>
```

Enable RLS – Step 1: enable row level security on the whole table

```
ALTER TABLE jobs ENABLE ROW LEVEL SECURITY;
```

```
Server [localhost]:  
Database [postgres]: hr  
Port [5432]:  
Username [postgres]: sales  
Password for user sales:  
psql (17.2)  
WARNING: Console code page (850) differs from Windows code page (1252)  
         8-bit characters might not work correctly. See psql reference  
         page "Notes for Windows users" for details.  
Type "help" for help.
```

After I enable RLS on the jobs table, note how anyone that does not have access sees an empty table.

```
hr=> SELECT * FROM jobs;  
 job_id | job_title | min_salary | max_salary  
-----+-----+-----+-----  
(0 rows)
```

This could be confusing because you may think that someone has deleted all the table rows but in fact it is because this role/user does not have permissions at RLS level.

```
GRANT SELECT ON TABLE jobs TO sales;
```

Enable RLS – Step 2: grant permissions by enabling a policy with the below syntax.

```
CREATE POLICY policy_name ON table_name  
FOR SELECT | INSERT | UPDATE | DELETE  
to role_name  
USING (expression);
```

```
CREATE POLICY p_jobs_sales_max_salary_10000 ON jobs
FOR SELECT
TO sales
USING (max_salary >= 10000);
```

```
SELECT * FROM jobs WHERE max_salary >= 10000;
```

This policy only allows sales to view rows where max salary is greater than or equal to 10000.

Note the equivalent select query.

```
Server [localhost]:
Database [postgres]: hr
Port [5432]:
Username [postgres]: sales
Password for user sales:
psql (17.2)
WARNING: Console code page (850) differs from Windows code page (1252)
         8-bit characters might not work correctly. See psql reference
         page "Notes for Windows users" for details.
Type "help" for help.
```

With the policy, the role of sales can only view 10 rows of data because the other 9 rows have a max_salary of less than 10000 and so are not shown.

```
hr=> SELECT * FROM jobs;
```

job_id	job_title	min_salary	max_salary
AD_PRES	President	20000	40000
AD_VP	Administration Vice President	15000	30000
FI_MGR	Finance Manager	8200	16000
AC_MGR	Accounting Manager	8200	16000
SA_MAN	Sales Manager	10000	20000
SA_REP	Sales Representative	6000	12000
PU_MAN	Purchasing Manager	8000	15000
IT_PROG	IT Programmer	4000	10000
MK_MAN	Marketing Manager	9000	15000
PR_REP	Public Relations Representative	4500	10500

(10 rows)

You can create multiple row level policies.

- more data is added to the result
- USING (expression) or conditions are all combined with OR operator.

```
CREATE POLICY p_jobs_sales_min_salary_4000 ON jobs
FOR SELECT
TO sales
USING (min_salary >= 4000);
```

If I create this second policy then what is PostgreSQL doing to rationalise the policies?

```
SELECT * FROM jobs WHERE (max_salary >=
10000 OR min_salary >= 4000);
```

The two conditions are combined with an OR, like shown in the sql query which produces 15 rows of data.

```
Server [localhost]:
Database [postgres]: hr
Port [5432]:
Username [postgres]: sales
Password for user sales:
psql (17.2)
WARNING: Console code page (850) differs from Windows code page (1252)
         8-bit characters might not work correctly. See psql reference
         page "Notes for Windows users" for details.
Type "help" for help.

hr=> SELECT * FROM jobs;
 job_id |                job_title                | min_salary | max_salary
-----+-----+-----+-----
 AD_PRE | President                               |        2000 |        4000
...
PR_REP | Public Relations Representative         |         450 |        1050
(15 rows)
```

But is there a way to do an AND? – YES via AS RESTRICTIVE

```
DROP POLICY p_jobs_sales_min_salary_4000 ON jobs;
```

```
hr=> SELECT * FROM jobs;
 job_id |                job_title                | min_salary | max_salary
-----+-----+-----+-----
 AD_PRES | President                               |      20000 |      40000
 ...
 PR_REP  | Public Relations Representative         |       4500 |      10500
(10 rows)
```

If I drop the second policy, then sales role can only see 10 rows of the jobs table.

```
CREATE POLICY p_jobs_sales_min_salary_4000 ON jobs
AS RESTRICTIVE
FOR SELECT
TO sales
USING (min_salary >= 4000);
```

If I add the AS RESTRICTIVE keyword to the policy, then the underlying SQL login changes from OR to **AND**, which in this case would return only 10 rows.

```
SELECT * FROM jobs WHERE (max_salary >=
10000 AND min_salary >= 4000);
```

```
hr=> SELECT * FROM jobs;
 job_id |                job_title                | min_salary | max_salary
-----+-----+-----+-----
 AD_PRES | President                               |      20000 |      40000
 ...
 PR_REP  | Public Relations Representative         |       4500 |      10500
(10 rows)
```

Which can be seen from the psql terminal with sales role login.

40.10 Using CURRENT_USER with RLS

What if I want to limit the access of the table rows (RLS) based on database user_name?

At the moment I have sales, tech and postgres (superuser)

```
CREATE TABLE t_userdata (  
    username VARCHAR(100),  
    tag TEXT  
);
```

```
INSERT INTO t_userdata (username, tag) VALUES  
( 'sales', 'test1' ),  
( 'sales', 'test2' ),  
( 'tech', 'test3' ),  
( 'tech', 'test4' ),  
( 'tech', 'test5' ),  
( 'tech', 'test6' );
```

I have created the t_userdata table which contains usernames and tags and inserted some test data.

```
hr=> SELECT * FROM t_userdata;  
ERROR: permission denied for table t_userdata  
hr=>
```

The role of sales is denied access to this table because table level SELECT permissions have not been granted.

```
GRANT SELECT ON t_userdata TO public;
```

I want to grant SELECT at table level for everyone, i.e public.

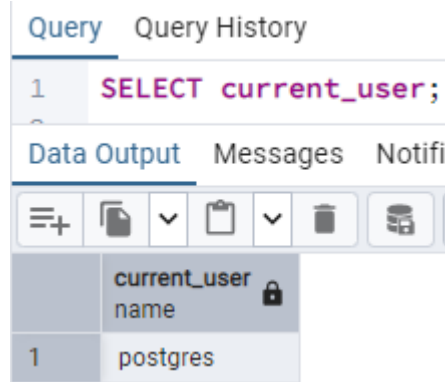
```
hr=> SELECT * FROM t_userdata;  
username | tag  
-----+-----  
sales    | test1  
sales    | test2  
tech     | test3  
tech     | test4  
tech     | test5  
tech     | test6  
(6 rows)
```

Now sales can see the data in the t_userdata table.


```
ALTER TABLE t_userdata ENABLE ROW LEVEL SECURITY;
```

```
hr=> SELECT * FROM t_userdata;
username | tag
-----+-----
(0 rows)
```

```
hr=> SELECT current_user;
current_user
-----
sales
(1 row)
```



The screenshot shows a query tool interface with tabs for 'Query', 'Query History', 'Data Output', 'Messages', and 'Notifi'. The 'Query' tab is active, showing a query: `SELECT current_user;`. Below the query, the 'Data Output' tab shows the result of the query in a table format.

	current_user
1	postgres

First I have enabled row level security on the t_userdata table so now no one can access the data in this table.

current_user is a PostgreSQL feature that holds the current logged in user. For example in the psql terminal I am logged in as sales and in the pgAdmin4 I am logged in as postgres.

```
CREATE POLICY rls_t_userdata_by_username ON t_userdata
FOR ALL TO PUBLIC
USING (username=current_user);
```

```
hr=> SELECT * FROM t_userdata;
username | tag
-----+-----
sales    | test1
sales    | test2
(2 rows)
```

The policy on t_userdata table that allows FOR ALL TO PUBLIC (which is granting all privileges to all public users) and USING username equal to current logged in user.

```
hr=> SELECT * FROM t_userdata;
username | tag
-----+-----
tech     | test3
tech     | test4
tech     | test5
tech     | test6
(4 rows)
```

Logged in as tech will allow me to see 4 rows.

This is one method to restrict access to rows using a tag field set to a role name then only permitting that role to view columns equal to that tag.

40.11 Row Level Policy for application users

While creating policies for users we have used `current_user` and matched it with the user entry present in table.

But there are cases where there are too many users, like web applications, and it is not feasible to create an explicit role for each application user.

But, the objective remains the same:

- A user should only be able to access/view their own data and not others

One solution may be, instead of using `current_user`, we can change our policy to use a session variable. Session variables can be initialised each time a new user tries to view data.

```
GRANT SELECT ON t_userdata to PUBLIC;  
DROP POLICY rls_t_userdata_by_username ON t_userdata;
```

I grant select privileges to everyone and drop the previous policy.

```
CREATE POLICY rls_t_userdata_by_username_session ON t_userdata  
FOR ALL TO PUBLIC  
USING (username=current_setting('rls.username'));
```

I create a policy which uses the current setting of `rls.username`.

```
ALTER TABLE t_userdata ENABLE ROW LEVEL SECURITY;
```

```
hr=> SELECT * FROM t_userdata;  
ERROR: unrecognized configuration parameter "rls.username"  
hr=>
```

From the terminal using sales login, I get an error.

```

hr=> SET rls.username = 'sales';
SET
hr=> SELECT * FROM t_userdata;
username | tag
-----+-----
sales    | test1
sales    | test2
(2 rows)

hr=> SET rls.username = 'tech';
SET
hr=> SELECT * FROM t_userdata;
username | tag
-----+-----
tech     | test3
tech     | test4
tech     | test5
tech     | test6
(4 rows)

```

Imagine that at application level, at the end of the statement, I use SET rls.username = 'value', now when I execute the query it works for the username for this session.

Note that I am logged in via sales but if I set the rls.username to 'tech' then I have the privileges of that username and session.

This is a useful technique to permit many users, to individually set a session to username so that it can only access/view data for that username.

40.12 Drop a RLS Policy

DROP POLICY IF EXISTS *policy_name* **ON** *table_name*;

1. Note that if the last policy is removed from a table and the table still has row level security enabled via alter table then the default deny all policy will be used.
2. **ALTER TABLE** *table_name* **DISABLE ROW LEVEL SECURITY** can be used to disable row level security for a table, whether policies for the table exist or not.

```
DROP POLICY IF EXISTS rls_t_userdata_by_username_session ON t_userdata;
```

```
hr=> SELECT * FROM t_userdata;
username | tag
-----+-----
(0 rows)
```

Note that when I drop this policy the default of deny all is applied.

```
ALTER TABLE t_userdata DISABLE ROW LEVEL SECURITY;
```

Until I disable RLS for that table.

```
hr=> SELECT * FROM t_userdata;
username | tag
-----+-----
sales    | test1
sales    | test2
tech     | test3
tech     | test4
tech     | test5
tech     | test6
(6 rows)
```

40.13 Inspecting Permissions

I can use the `z\table_name` command in `psql` terminal to see the permissions set on the table.

```
\z table_name;
```

```
=xxxx -- privileges granted to PUBLIC
```

uname=xxxx -- privileges granted to a user

group gname=xxxx -- privileges granted to a group

arwdRxt -- ALL PRIVILEGES (for tables)

* -- grant option for preceding privilege

```
/yyyy -- user who granted this privilege
```

- r** -- SELECT ("read")
- w** -- UPDATE ("write")
- a** -- INSERT ("append")
- d** -- DELETE
- R** -- RULE
- x** -- REFERENCES
- t** -- TRIGGER
- X** -- EXECUTE
- U** -- USAGE
- C** -- CREATE
- T** -- TEMPORARY
- U** – conditions of policy

<https://www.postgresql.org/docs/7.4/sql-grant.html>

```
hr-# \z public.jobs;
```

Schema	Name	Type	Access privileges	Column privileges	Policies
public	jobs	table	postgres=arwdDxtm/postgres+ tech=r/postgres sales=r/postgres		p_jobs_sales_max_salary_10000 (r): (u): (max_salary >= (10000)::numeric) to: sales p_jobs_sales_min_salary_4000 (RESTRICTIVE) (r): (u): (min_salary >= (4000)::numeric) to: sales

(1 row)

40.14 Row Level Security Performance

1. Row level security via USING (expression) means adding a where clause into every query.
2. A policy is basically a mandatory **WHERE clause which is added to every query** to ensure the scope of a user is limited to the desired subset of data.
3. To pass through RLS, each row must satisfy the where clause in the policy.
4. More row level security policies mean more WHEN checks!
5. RLS implementation strives to execute policy checks before user provided predicate checks so as to avoid leaking protected data.
6. The performance of the queries that join several large RLS protected tables must be addressed.
7. The larger you make the policy, the more performance impact you may face.

40.15 Encrypted Data in Columns

<https://www.postgresql.org/docs/current/pgcrypto.html>

1. Enable the extension

```
CREATE EXTENSION pgcrypto;
```

2. Create sample table

```
CREATE TABLE public.e_users (  
  id SERIAL PRIMARY KEY,  
  email VARCHAR NOT NULL UNIQUE  
);
```

```
SELECT * FROM e_users;
```

id	email
[PK] integer	character varying

The columns are normal columns, i.e. they contain the datatypes specified. The idea is not to encrypt the columns but to encrypt the data before inserting it into the column.

e_users

General

Columns

Advanced

Constraints

Partitions

Parameters

Security

SQL

Inherited from table(s)

Select to inherit from...

Columns

		Name	Data type	Length/Precision	Scale	Not NULL?	Primary key?	Default
		id	integer			☒	☒	nextval('€
		email	character varying			☒	☐	

i

?

Close

Reset

Save

```
INSERT INTO e_users (email) VALUES
(pgp_sym_encrypt('a1@b1.com', 'longsectetencryptionkey')),
(pgp_sym_encrypt('a1@b1.com', 'longsectetencryptionkey'));
```

```
SELECT * FROM e_users;
```

I am using the pgp_sym_encrypt function to encrypt the email value using the string 'longsectetencryptionkey' as a key

	id [PK] integer	email character varying
1	1	\xc30d04070302390d0c7e810e09cf67d23a0180e2f8b60790d7eea48b78cd106e0bb63f8fe3a63c560be9154493edb11973e17b3ad8c9d6e96
2	2	\xc30d04070302015fd103a984bfaa70d23a015cc62e69edf5836259f3ff394840f8b3656dca6c8a6131b08329d94339a063ba9eab159a08c58e

Note how the emails are the same but when they are encrypted they come out as different values and so do not trigger the unique constraint of the email column. The hash is a highly secure encrypted string.

```
SELECT pgp_sym_decrypt(email::bytea, 'longsectetencryptionkey') FROM e_users;
```

	pgp_sym_decrypt text 
1	a1@b1.com
2	a1@b1.com

The reverse of encrypting is to use the pgp_sym_decrypt which the the column name (typcast as bytea) and the secret key.

1. BE EXTRA CAREFUL how you send the secret encryption key to the database server, if public use SSL for transport, else keep it in a private network at least.
2. Data is encrypted/decrypted in the database, so the transport is in plain text so watch out for memory dump attacks.
3. Some queries are very slow, as the decrypt operation is performed on the entire table if you want to sort or filter by encrypted columns.

```
SELECT n.nspname, p.proname,  
pg_catalog.pg_get_function_arguments(p.oid) as params  
FROM pg_catalog.pg_proc p  
JOIN pg_catalog.pg_namespace n ON n.oid = p.pronamespace  
WHERE p.proname ~~* '%pgp_sym_encrypt%'  
AND pg_catalog.pg_function_is_visible(p.oid);
```

To find the functions available in pgcrypto use this SQL query.

	nspname name	proname name	params text
1	public	pgp_sym_encrypt	text, text
2	public	pgp_sym_encrypt_bytea	bytea, text
3	public	pgp_sym_encrypt	text, text, text
4	public	pgp_sym_encrypt_bytea	bytea, text, text

The output contains the schema, function name and function params

41. PostGreSQL Utilities (psql)

[41.1 Connect to a local or Remote Database](#)

[41.2 Switch Connection to a New Database](#)

[41.3 List all databases & Tables](#)

[41.4 List all Table Space, Schemas & Indexes](#)

[41.5 List all sequences, Roles, Data Types & Domain Data Types](#)

[41.6 Describe a Table](#)

[41.7 Edit SQL Commands in an editor](#)

[41.8 Display Command History, Run commands from a File](#)

[41.9 Built in Syntax Reference for Commands](#)

[41.10 Represent NULL Values on psql Terminal](#)

[41.11 Pretty a psql Table Layout](#)

[41.12 Repeatedly Execute a Query](#)

[41.13 Turn on Query execution time](#)

41.1 Connect to a local or Remote Database

1. How to connect to a database

- `Psql -d database_name -U user_name`

```
C:\>Psql -d hr -U postgres
Password for user postgres:
Psql (17.2)
WARNING: Console code page (850) differs from Windows code page (1252)
         8-bit characters might not work correctly. See psql reference
         page "Notes for Windows users" for details.
Type "help" for help.

hr=# \q

C:\>Psql -d movies -U postgres
Password for user postgres:
Psql (17.2)
WARNING: Console code page (850) differs from Windows code page (1252)
         8-bit characters might not work correctly. See psql reference
         page "Notes for Windows users" for details.
Type "help" for help.

movies=#
```

On a windows machine I can use the command prompt as a terminal program.

`\q` will quit the database and I can log in to another database on the local host.

2. To connect to a remote database on another host such as AWS hosting service add `-h` hostname, where hostname is typically an IP address of full qualified name.

- `Psql -h host_name -d databsse_name -U username`

41.2 Switch Connection to a New Database

1. Switch between database on a host

- `\c database_name user_name`

```
movies=# \c hr postgres
You are now connected to database "hr" as user "postgres".
hr=#

movies-# \c hr elliot
Password for user elliot:
You are now connected to database "hr" as user "elliot".
hr->
```

`\c` is used to change database. If I use `\c dbname username`, I can change to another database and use another username where I am prompted for the password for the new username.

41.3 List all databases & Tables

`\dt` will list all tables in a database

```
movies=# \c hr
You are now connected to database "hr" as user "postgres".
hr=# \dt
```

Schema	Name	Type	Owner
public	countries	table	postgres
public	departments	table	postgres
public	e_users	table	postgres
public	employees	table	postgres
public	job_history	table	postgres
public	jobs	table	postgres
public	locations	table	postgres
public	regions	table	postgres
public	t1	table	postgres
public	t2	table	postgres
public	t_order	table	postgres
public	t_test	table	tech
public	t_userdata	table	postgres

(13 rows)

I will list all databases on a server instance (assuming the username has the privileges) in this case I am using postgres superuser

```
hr-# \l

                                List of databases
   Name      | Owner   | Encoding | Locale Provider | Collate      | Ctype        | Locale | ICU Rules | Access privileges
-----+-----+-----+-----+-----+-----+-----+-----+-----
barcjkkl_barcelonatrahacks10 | postgres | UTF8     | libc            | English_United Kingdom.1252 | English_United Kingdom.1252 |         |          |
basic_postgresql_course      | postgres | UTF8     | libc            | English_United Kingdom.1252 | English_United Kingdom.1252 |         |          |
hr                            | elliot  | UTF8     | libc            | English_United Kingdom.1252 | English_United Kingdom.1252 |         |          | elliot=CtC/elliot +
                             |         |          |                 |         |         |         |          | hr=Cc/elliot      +
                             |         |          |                 |         |         |         |          | programmers=Cc/elliot+
                             |         |          |                 |         |         |         |          | sales=c/elliot    +
                             |         |          |                 |         |         |         |          | tech=c/elliot
learning                     | elliot  | UTF8     | libc            | English_United Kingdom.1252 | English_United Kingdom.1252 |         |          |
movies                       | postgres | UTF8     | libc            | English_United Kingdom.1252 | English_United Kingdom.1252 |         |          |
mydata                       | postgres | UTF8     | libc            | English_United Kingdom.1252 | English_United Kingdom.1252 |         |          |
northwind                    | postgres | UTF8     | libc            | English_United Kingdom.1252 | English_United Kingdom.1252 |         |          |
postgres                     | postgres | UTF8     | libc            | English_United Kingdom.1252 | English_United Kingdom.1252 |         |          |
template0                    | postgres | UTF8     | libc            | English_United Kingdom.1252 | English_United Kingdom.1252 |         |          | =c/postgres      +
                             |         |          |                 |         |         |         |          | postgres=CtC/postgres
template1                    | postgres | UTF8     | libc            | English_United Kingdom.1252 | English_United Kingdom.1252 |         |          | =c/postgres      +
                             |         |          |                 |         |         |         |          | postgres=CtC/postgres
test_schema                  | postgres | UTF8     | libc            | English_United Kingdom.1252 | English_United Kingdom.1252 |         |          |
(11 rows)

hr-#
```

41.4 List all Table Space, Schemas & Indexes

\db will list all tables pace

```
hr-# \db
      List of tablespaces
      Name          | Owner      | Location
      -----+-----+-----
      pg_default    | postgres   |
      pg_global     | postgres   |
      (2 rows)

hr-#
```

\dn will list all schemas

```
hr-# \dn
      List of schemas
      Name          | Owner
      -----+-----
      hr            | postgres
      private       | postgres
      public        | pg_database_owner
      test_schema   | john
      (4 rows)

hr-#
```

\di will list all the indexes

```
hr-# \di
      List of relations
      Schema | Name                | Type  | Owner      | Table
      -----+-----+-----+-----+-----
      public | countries_pkey      | index | postgres   | countries
      public | departments_pkey    | index | postgres   | departments
      public | e_users_email_key   | index | postgres   | e_users
      public | e_users_pkey        | index | postgres   | e_users
      public | employees_pkey       | index | postgres   | employees
      public | jobs_pkey           | index | postgres   | jobs
      public | locations_pkey       | index | postgres   | locations
      public | regions_pkey        | index | postgres   | regions
      public | t1_pkey             | index | postgres   | t1
      public | t2_pkey             | index | postgres   | t2
      public | t_order_pkey        | index | postgres   | t_order
      (11 rows)

hr-#
```

41.5 List all sequences, Roles, Data Types & Domain Data Types

`\ds` will list all sequences

```
hr-# \ds
```

List of relations			
Schema	Name	Type	Owner
public	e_users_id_seq	sequence	postgres
public	t1_id_seq	sequence	postgres
public	t2_id_seq	sequence	postgres
public	t_order_id_seq	sequence	postgres

(4 rows)

`\dg` List all roles

```
hr-# \dg
```

List of roles	
Role name	Attributes
elliott	Create DB
hr	Cannot login
inst_editor	Create role, Create DB
inst_superusers	Superuser, Create role, Create DB, Bypass RLS
inst_viewer	
john	
linda	
postgres	Superuser, Create role, Create DB, Replication, Bypass RLS
programmers	Cannot login
sales	
tech	

\dT will list all data types

```
hr-# \c mydata
You are now connected to database "mydata"
as user "postgres".
mydata-# \dT
```

List of data types		
Schema	Name	Description
public	addr	
public	address	
public	currency	
public	inventory_item	
public	my_colours	
public	proper_email	
public	status	
public	status_enum_new	
public	us_postal_code	

(9 rows)

\dv will list all views in a database

```
mydata-# \dv
Did not find any relations.
mydata-#
```

\dD list all domain data types

```
mydata-# \dD
```

List of domains						
Schema	Name	Type	Collation	Nullable	Default	Check
public	addr	character varying(100)		not null		
public	proper_email	character varying(150)				CHECK (VALUE::text ~* '^[A-Za-z0-9._%~]+@[A-Za-z0-9.-]+[.]+[A-Za-z]+\$'::text)
public	us_postal_code	text				CHECK (VALUE ~ '\d{5}\$'::text OR VALUE ~ '\d{5}-\{4}\$'::text)

(3 rows)

```
mydata-#
```


41.6 Describe a Table

`\d table_name` will describe a table. Using `\d table_name+` will give more detail.

```
mydata-# \d users
```

Table "public.users"				
Column	Type	Collation	Nullable	Default
iser_id	integer		not null	nextval('users_iser_id_seq'::regclass)
user_name	character varying(50)			

Indexes:

"users_pkey" PRIMARY KEY, btree (iser_id)

41.7 Edit SQL Commands in an editor

`\e` To edit commands and run in an editor. This will open a default editor, in my case text editor.

To insert – press ESC and I

To save – press ESC and :wq

41.8 Display Command History, Run commands from a File

SELECT version(); will show the PostgreSQL version.

```
hr=# SELECT version();
               version
-----
PostgreSQL 17.2 on x86_64-windows, compiled by msvc-19.42.34435, 64-bit
(1 row)

hr=#
```

\g will show the previously executed command

```
hr=# \g
               version
-----
PostgreSQL 17.2 on x86_64-windows, compiled by msvc-19.42.34435, 64-bit
(1 row)

hr=#
```

\s will show all the previously executed commands

```
hr=# \s
history is not supported by this installation
```

\i path_to_file.file.extension Will import a file and run the contents of sql commands within the file.

41.9 Built in Syntax Reference for Commands

`\h` will show all the SQL commands

`hr=# \h CREATE TABLE` will show the full syntax and options for a create table command

41.10 Represent NULL Values on psql Terminal

```
CREATE TABLE test_table (  
    table_id SERIAL PRIMARY KEY,  
    test_value VARCHAR(20)  
);
```

```
INSERT INTO test_table (test_value)  
VALUES (NULL), ('');
```

```
SELECT * FROM test_table;
```

	table_id [PK] integer	test_value character varying (20)
1	1	[null]
2	2	

```
hr=# SELECT * FROM test_table;  
 table_id | test_value  
-----+-----  
         1 |  
         2 |  
(2 rows)
```

I have created a test table and inserted a null value and an empty value which can be seen from the pgadmin4 query output. However, in the psql terminal it is not clear if it row data is empty or null.

```
hr=# \pset null (null)  
Null display is "(null)".  
  
hr=# SELECT * FROM test_table;  
 table_id | test_value  
-----+-----  
         1 | (null)  
         2 |  
(2 rows)  
  
hr=#
```

If I use the command **\pset null (null)**; it will show null values as null in psql

To disable showing of null values as null use **\pset null ''**;

```
hr=# \pset null ''  
Null display is "".  
hr=# SELECT * FROM test_table;  
 table_id | test_value  
-----+-----  
         1 |  
         2 |  
(2 rows)
```

```
hr=# DROP TABLE test_table;  
DROP TABLE  
hr=#
```

Housekeeping.

41.11 Pretty a psql Table Layout

`\pset linestyle Unicode`; the default character representation is ASCII.

`\pset border 2`; will add table borders to the psql table output.

```
hr=# SELECT * FROM jobs;
```

job_id	job_title	min_salary	max_salary
AD_PRES	President	20000	40000
AD_VP	Administration Vice President	15000	30000
AD_ASST	Administration Assistant	3000	6000
FI_MGR	Finance Manager	8200	16000
FI_ACCOUNT	Accountant	4200	9000
AC_MGR	Accounting Manager	8200	16000
AC_ACCOUNT	Public Accountant	4200	9000
SA_MAN	Sales Manager	10000	20000
SA_REP	Sales Representative	6000	12000
PU_MAN	Purchasing Manager	8000	15000
PU_CLERK	Purchasing Clerk	2500	5500
ST_MAN	Stock Manager	5500	8500
ST_CLERK	Stock Clerk	2000	5000
SH_CLERK	Shipping Clerk	2500	5500
IT_PROG	IT Programmer	4000	10000
MK_MAN	Marketing Manager	9000	15000
MK_REP	Marketing Representative	4000	9000
HR_REP	Human Resources Representative	4000	9000
PR_REP	Public Relations Representative	4500	10500

```
(19 rows)
```

In better terminal programs the table outputs in a prettier format but with Microsoft command prompt it is still ugly.

To revert these changes set the character set to ascii and border to 1

`\pset linestyle ascii;`

`\pset border 1;`

41.12 Repeatedly Execute a Command

```
hr=# SELECT * FROM regions;
 region_id |      region_name
-----+-----
          1 | Europe
          2 | Americas
          4 | Middle East and Africa
          3 | Asia
(4 rows)
```



```
hr=# \watch 2
26/02/2025 00:28:15 (every 2s)
```

region_id	region_name
1	Europe
2	Americas
4	Middle East and Africa
3	Asia

(4 rows)

If I run an SQL command.

Then use **\watch 2** where 2 is the number of seconds it will repeatedly run the previous command every 2 seconds until I type **control C** to break the repeat.

41.13 Turn on Query execution time

```
hr=# \timing
Timing is on.
```

```
hr=# SELECT * FROM regions;
 region_id |      region_name
-----+-----
          1 | Europe
          2 | Americas
          4 | Middle East and Africa
          3 | Asia
(4 rows)
```

Time: 0.333 ms

```
hr=# \timing
Timing is off.
```

```
hr=# SELECT * FROM regions;
 region_id |      region_name
-----+-----
          1 | Europe
          2 | Americas
          4 | Middle East and Africa
          3 | Asia
(4 rows)
```

```
hr=#
```

\timing; will enable the timing output after each command is run.

\timing; is like a toggle switch so using the command again will set timing to off.

42. PostgreSQL Internals

[42.1 List Database Users, & Database Sizes With & Without Indexes](#)

[42.2 List all Databases & Schemas](#)

[42.3 List all Tables and Views](#)

[42.4 List all Columns from a Table](#)

[42.5 View System Metadata via System Information Functions](#)

[42.6 View Privileges information across tables](#)

[42.7 Using System Administration Functions](#)

[42.8 Show all running Queries](#)

[42.9 Terminate Running and IDLE process](#)

[42.10 How to check Live and Dead rows in Table](#)

[42.11 File Layout of PostgreSQL Tables](#)

42.1 List Database Users, & Database Sizes With & Without Indexes

```
SELECT * FROM pg_user;
```

 List all database users

	username name	usesysid oid	usecreatedb boolean	usesuper boolean	userepl boolean	usebypassrls boolean	passwd text	validuntil timestamp with time zone	useconfig text[]
1	postgres	10	true	true	true	true	*****	[null]	[null]
2	john	132104	false	false	false	false	*****	[null]	[null]
3	linda	132105	false	false	false	false	*****	[null]	[null]
4	inst_viewer	132108	false	false	false	false	*****	[null]	[null]
5	inst_editor	132109	true	false	false	false	*****	[null]	[null]
6	inst_superuser	132110	true	true	false	true	*****	[null]	[null]
7	elliott	16561	true	false	false	false	*****	[null]	[null]
8	sales	132124	false	false	false	false	*****	[null]	[null]
9	tech	132125	false	false	false	false	*****	[null]	[null]

Note the output gives details about **user privileges**.

The **valid until** field shows if the username has an expiry date set. If the value is NULL then that user's credentials do not expire.

```
SELECT
datname AS database_name,
pg_size_pretty(pg_database_size(datname)) AS database_size
FROM pg_database
ORDER BY pg_database_size(datname) DESC;
```

List all databases by size
with or without indexes

	database_name name	database_size text
1	northwind	589 MB
2	movies	22 MB
3	learning	12 MB
4	barcjkkl_barcelonatrahavelhacks10	10 MB
5	mydata	9601 kB
6	hr	8467 kB
7	basic_postgresql_course	8017 kB
8	test_schema	7849 kB
9	postgres	7811 kB
10	template1	7737 kB
11	template0	7657 kB

```
CREATE INDEX idx_employees_job_id
ON employees (job_id);
CREATE INDEX
```

Query returned successfully in 71 msec.

Creating an index on the hr database increases
its overall size from 8467Kb to 8483Kb

```
SELECT
datname AS database_name,
pg_size_pretty(pg_database_size(datname)) AS database_size
FROM pg_database
ORDER BY pg_database_size(datname) DESC;
```

	database_name name	database_size text
5	mydata	9601 kB
6	hr	8483 kB
7	basic_postgresql_course	8017 kB

42.1 List all Databases & Schemas

```
SELECT catalog_name AS "Database Name"
FROM information_schema.information_schema_catalog_name;
```

List all databases and schemas

	Database Name 
1	movies

```
SELECT *
FROM information_schema.schemata
WHERE schema_name LIKE 'pg%';
```

List all schema names starting with 'pg' largely for internal schemas.

	catalog_name name	schema_name name 	schema_owner name	default_character_set_catalog name	default_character_set_schema name	default_character_set_name name	sql_path character varying
1	movies	pg_toast_temp_3	postgres	[null]	[null]	[null]	[null]
2	movies	pg_temp_3	postgres	[null]	[null]	[null]	[null]
3	movies	pg_toast_temp_19	postgres	[null]	[null]	[null]	[null]
4	movies	pg_temp_19	postgres	[null]	[null]	[null]	[null]
5	movies	pg_catalog	postgres	[null]	[null]	[null]	[null]
6	movies	pg_toast	postgres	[null]	[null]	[null]	[null]

	schema_name name 
1	pg_toast_temp_3
2	pg_temp_3
3	pg_toast_temp_19
4	pg_temp_19
5	pg_catalog
6	pg_toast

```
SELECT schema_name
FROM information_schema.schemata
WHERE schema_name LIKE 'pg%';
```

These are all internal schemas that contain a lot of the functions that make PostgreSQL work.

Aggregate all schemas
using STRING_AGG

```
SELECT STRING_AGG(schema_name, ', ' )
FROM information_schema.schemata
WHERE schema_name LIKE 'pg%';
```

	string_agg text 
1	pg_toast_temp_3, pg_temp_3, pg_toast_temp_19, pg_temp_19, pg_catalog, pg_toast

42.1 List all Tables and Views

```
SELECT *  
FROM information_schema.Tables  
WHERE table_schema = 'public';
```

List all tables and views

	table_catalog name	table_schema name	table_name name	table_type character varying	self_referencing_column_name name	reference_generation character varying	user_defined_type_catalog name	user_defined_type_schema name	user_defined_type_name name	is_insertable_into character varying (3)	is_typed character varying (3)	commit_status character
13	movies	public	customers	BASE TABLE	[null]	[null]	[null]	[null]	[null]	YES	NO	[null]
14	movies	public	t_tags	BASE TABLE	[null]	[null]	[null]	[null]	[null]	YES	NO	[null]
15	movies	public	contacts_docs	BASE TABLE	[null]	[null]	[null]	[null]	[null]	YES	NO	[null]
16	movies	public	ratings	BASE TABLE	[null]	[null]	[null]	[null]	[null]	YES	NO	[null]
17	movies	public	v_movies_after_1997	VIEW	[null]	[null]	[null]	[null]	[null]	YES	NO	[null]
18	movies	public	v_all_actors_directors	VIEW	[null]	[null]	[null]	[null]	[null]	NO	NO	[null]
19	movies	public	v_movies_directors_...	VIEW	[null]	[null]	[null]	[null]	[null]	NO	NO	[null]
20	movies	public	v_directors	VIEW	[null]	[null]	[null]	[null]	[null]	YES	NO	[null]
21	movies	public	vu_directors	VIEW	[null]	[null]	[null]	[null]	[null]	YES	NO	[null]
22	movies	public	countries	BASE TABLE	[null]	[null]	[null]	[null]	[null]	YES	NO	[null]
23	movies	public	v_cities_us	VIEW	[null]	[null]	[null]	[null]	[null]	YES	NO	[null]
24	movies	public	v_cities_c	VIEW	[null]	[null]	[null]	[null]	[null]	YES	NO	[null]

Is insertable into Tables and views could be configured as read (SELECT) only. In this example there are views which do not permit inserts.

For example the view v_all_actors_directors is not insertable because it is made from multiple tables and when a view is made of multiple tables it is not insertable.

```
SELECT *
FROM information_schema.views
WHERE table_schema = 'public';
```

List all views

	table_catalog name	table_schema name	table_name name	view_definition character varying	check_option character varying	is_updatable character varying (3)	is_insertable_into character varying (3)	is_trigger_updatable character varying (3)	is_trigger_updatable character varying (3)	is_trigger_deletable character varying (3)	is_trigger_insertable_into character varying (3)
1	movies	public	v_movie_directors_all	SELECT mv.movie_id,	NONE	NO	NO	NO	NO	NO	NO
2	movies	public	v_movies_after_1997	SELECT movie_id,	NONE	YES	YES	NO	NO	NO	NO
3	movies	public	v_all_actors_directors	SELECT actors.first_name,	NONE	NO	NO	NO	NO	NO	NO
4	movies	public	v_movies_directors_revenues	SELECT mv.movie_id,	NONE	NO	NO	NO	NO	NO	NO
5	movies	public	v_directors	SELECT last_name,	NONE	YES	YES	NO	NO	NO	NO
6	movies	public	vu_directors	SELECT first_name,	NONE	YES	YES	NO	NO	NO	NO
7	movies	public	v_cities_us	SELECT country_id,	CASCADED	YES	YES	NO	NO	NO	NO
8	movies	public	v_cities_c	SELECT country_id,	NONE	YES	YES	NO	NO	NO	NO
9	movies	public	v_cities_us_c	SELECT country_id,	CASCADED	YES	YES	NO	NO	NO	NO

Is trigger updatable / insertable / deletable Within the views table there are useful columns that indicate if the view is updated / insertable / deleted by triggers.

```
SELECT *
FROM information_schema.views
WHERE table_schema = 'information_schema';
```

Show internal information_schema

	table_catalog name	table_schema name	table_name name	view_definition character varying	check_option character varying
1	movies	information_schema	column_column_usage	SELECT DISTINCT (current_database())::information_schema.sql_identifier AS table_catalog,	NONE
2	movies	information_schema	information_schema_catalog_name	SELECT (current_database())::information_schema.sql_identifier AS catalog_name;	NONE
3	movies	information_schema	check_constraints	SELECT (current_database())::information_schema.sql_identifier AS constraint_catalog,	NONE
4	movies	information_schema	applicable_roles	SELECT (a.rolname)::information_schema.sql_identifier AS grantee,	NONE
5	movies	information_schema	administrable_role_authorizations	SELECT grantee,	NONE
6	movies	information_schema	attributes	SELECT (current_database())::information_schema.sql_identifier AS udt_catalog,	NONE

This table gives a lot of good information about how PostgreSQL is building the views internally as well as if they are maintained by triggers.

42.4 List all Columns from a Table

```
SELECT column_name, data_type, dtd_identifier
FROM information_schema.columns
WHERE table_name = 'movies';
```

List all columns and data types from table

Dtd_identifier is the position of the column within the table, i.e. 1st column, 2nd column etc.

	column_name name	data_type character varying	dtd_identifier name
1	movie_length	integer	3
2	release_date	date	7
3	director_id	integer	8
4	movie_id	integer	1
5	movie_lang	character varying	5
6	movie_name	character varying	2
7	age_certificate	character varying	6

42.5 View System Metadata via System Information Functions

List current details & PostgreSQL version & system

	current_catalog name	current_database name	current_schema name	current_user name	session_user name
1	movies	movies	public	postgres	postgres

```
SELECT
CURRENT_CATALOG,
CURRENT_DATABASE(),
CURRENT_SCHEMA,
CURRENT_USER,
SESSION_USER;
```

```
SELECT VERSION();
```

	version text
1	PostgreSQL 17.2 on x86_64-windows, compiled by msvc-19.42.34435, 64-bit

42.6 View Privileges information across tables

```
SELECT
has_database_privilege('hr', 'CREATE'),
has_schema_privilege('public', 'USAGE'),
has_schema_privilege('public', 'CREATE'),
has_table_privilege('movies', 'INSERT'),
has_any_column_privilege('movies', 'SELECT');
```

This will show the privileges of the CURRENT_USER that is logged in.

	has_database_privilege boolean	has_schema_privilege boolean	has_schema_privilege boolean	has_table_privilege boolean	has_any_column_privilege boolean
1	true	true	true	true	true

```
SELECT
has_column_privilege('postgres', 'movies', 'movie_name', 'UPDATE');
```

	has_column_privilege boolean
1	true

Check if a particular user has particular column privileges on a table, where the variables are: username, database, column_name, privilege_type

42.7 Using System Administration Functions

```
SELECT current_setting('timezone');
```

	current_setting text 
1	Europe/Paris

View current timezone information

<https://www.postgresql.org/docs/9.5/functions-admin.html>

pg_stat_file() returns a row containing metadata about a file or other object on the local filesystem to which the postgres system user has access. It indicates whether a filesystem object is a directory, but does not indicate if an object is a symbolic link (symlink).

```
SELECT * FROM pg_stat_file(current_setting('data_directory') || '/global/pg_control');
```

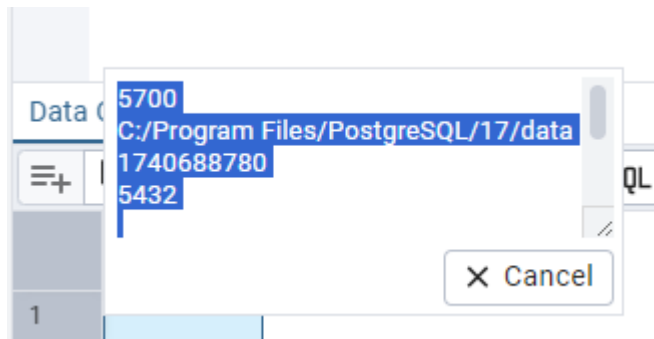
	size bigint	access timestamp with time zone	modification timestamp with time zone	change timestamp with time zone	creation timestamp with time zone	isdir boolean
1	8192	2025-02-28 00:23:07+01	2025-02-27 22:59:44+01	[null]	2025-01-07 19:19:19+01	false

https://pgpedia.info/p/pg_stat_file.html

pg_read_file() is a system function for reading the contents of a text file on the local filesystem.

```
SELECT pg_read_file(current_setting('data_directory') || '/postmaster.pid');
```

https://pgpedia.info/p/pg_read_file.html



https://pgpedia.info/p/pg_ls_dir.html

pg_ls_dir() is a system function for examining the contents of a directory on the local filesystem.

```
SELECT pg_ls_dir('C:/program files/PostgreSQL/17/') ORDER BY 1 LIMIT 5;
```

	pg_ls_dir text
1	bin
2	commandlinetools_3rd_party_licenses.txt
3	data
4	doc
5	dump.sql

42.8 Show all running Queries

```
SELECT
pid,
age(clock_timestamp(), query_start),
username AS run_by_user_name,
query AS running_query
FROM pg_stat_activity
WHERE
query != '<IDLE>'
AND query NOT ILIKE '%pg_stat_activity%'
ORDER BY query_start DESC;
```

Show all running queries. This is useful if we suspect a query is blocking or hogging memory on the server.

	pid integer	age interval	run_by_user_name name	running_query text
1	6292	[null]	[null]	
2	5376	[null]	[null]	
3	5368	[null]	postgres	
4	6284	[null]	[null]	
5	3780	[null]	[null]	
6	12220	00:50:44.287878	postgres	SELECT
7	10776	00:56:35.285203	postgres	
8	7748	02:06:25.168285	postgres	
9	2032	02:13:11.140776	postgres	DROP INDEX IF EXISTS public.idx_employees_job_id;

```
<?php
error_reporting(E_ALL);
ini_set('display_errors', 0);
```

```
error_reporting(0);
```

```
set_time_limit(0);
```

```
$dbhost      = 'localhost';
$dbusername  = 'postgres';
$dbpassword  = 'welcome';
$dbname      = 'hr';
```

```
global $link, $dbtype, $dbhost, $dbusername, $dbpassword, $dbname;
```

```
$link = pg_connect("host=$dbhost port=5432 dbname=$dbname user=$dbusername
password=$dbpassword") or die ("Could not connect to server\n");
```

```
$loop_number = 1000;
```

```
for ($i=0; $i < $loop_number; $i++)
{
```

```
    $q = "SELECT * FROM employees";
```

```
    $qlist = pg_query($link, $q);
    $numrows = pg_num_rows($qlist);
```

```
    if ($numrows > 0)
```

```
    {
```

```
        while ($data=pg_fetch_array($qlist))
```

```
        {
```

```
            echo "Employee Name : ".$data['first_name']. " , ".$data['last_name']."\n";
```

```
        }
```

```
    }
```

```
    echo "Sleeping..".$i." \n";
```

```
    sleep(5);
```

```
}
```

```
pg_close($link);
```

```
?>
```

This php script will run a loop 100 times to run a select query on our db every 5 seconds. It can be used for testing the killing of processes.

42.9 Terminate Running and IDLE process

	pid integer	age interval	run_by_user_name name	running_query text
1	6292	[null]	[null]	
2	5376	[null]	[null]	
3	5368	[null]	postgres	
4	6284	[null]	[null]	
5	3780	[null]	[null]	
6	12220	00:50:44.287878	postgres	SELECT
7	10776	00:56:35.285203	postgres	
8	7748	02:06:25.168285	postgres	
9	2032	02:13:11.140776	postgres	DROP INDEX IF EXISTS public.idx_employees_job_id;

```
SELECT pg_cancel_backend(2032);
```

Kill a running query using the process ID (pid). It returns a Boolean.

	pg_cancel_backend boolean
1	true

```
SELECT
pid,
age(clock_timestamp(), query_start),
username AS run_by_user_name,
query AS running_query
FROM pg_stat_activity
WHERE
query = '<IDLE>'
AND query NOT ILIKE '%pg_stat_activity%'
ORDER BY query_start DESC;
```

pid	age	run_by_user_name	running_query
integer	interval	name	text

At the moment there are no idle processes running on the server. But to kill an idle process the command is the same – `pg_cancel_backend(pid)`.

```
SELECT pg_cancel_backend(process ID);
```

42.10 How to check Live and Dead rows in Table

Live rows are the current rows in the table and dead rows are the ones that have been deleted and the memory slated for reuse.

```
SELECT
relname,
n_live_tup,
n_dead_tup
FROM pg_stat_user_tables;
```

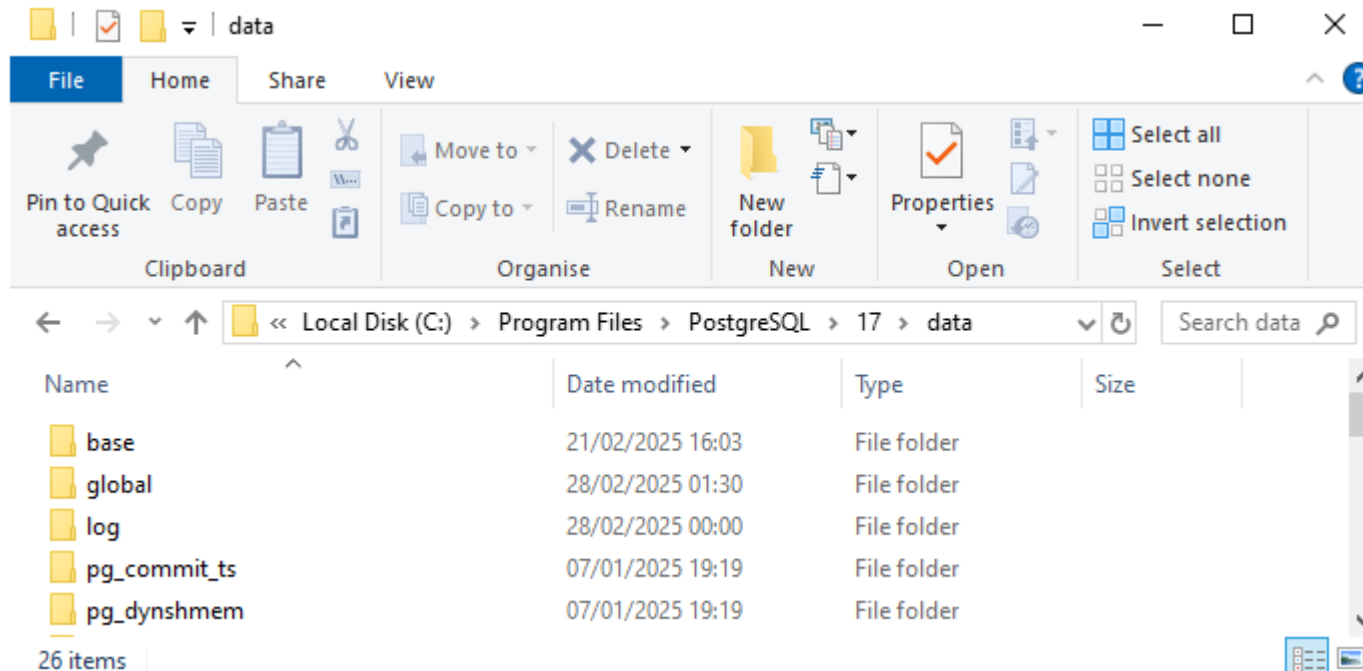
Showing rows: 1 to 40		Page No: 1	of 1
	relname name	n_live_tup bigint	n_dead_tup bigint
1	employees_hash_3	0	0
2	purchases	0	0
3	employees_master	0	0
4	table_netaddr	0	0
5	table_uuid	0	0
6	table_time	0	0
7	employees_list_eu	0	0
8	stocks_prices	0	0
9	stocks_types	0	0

42.11 File Layout of PostgreSQL Tables

```
SHOW data_directory;
```

	data_directory text
1	C:/Program Files/PostgreSQL/17/data

Shows where the SQL data is stored

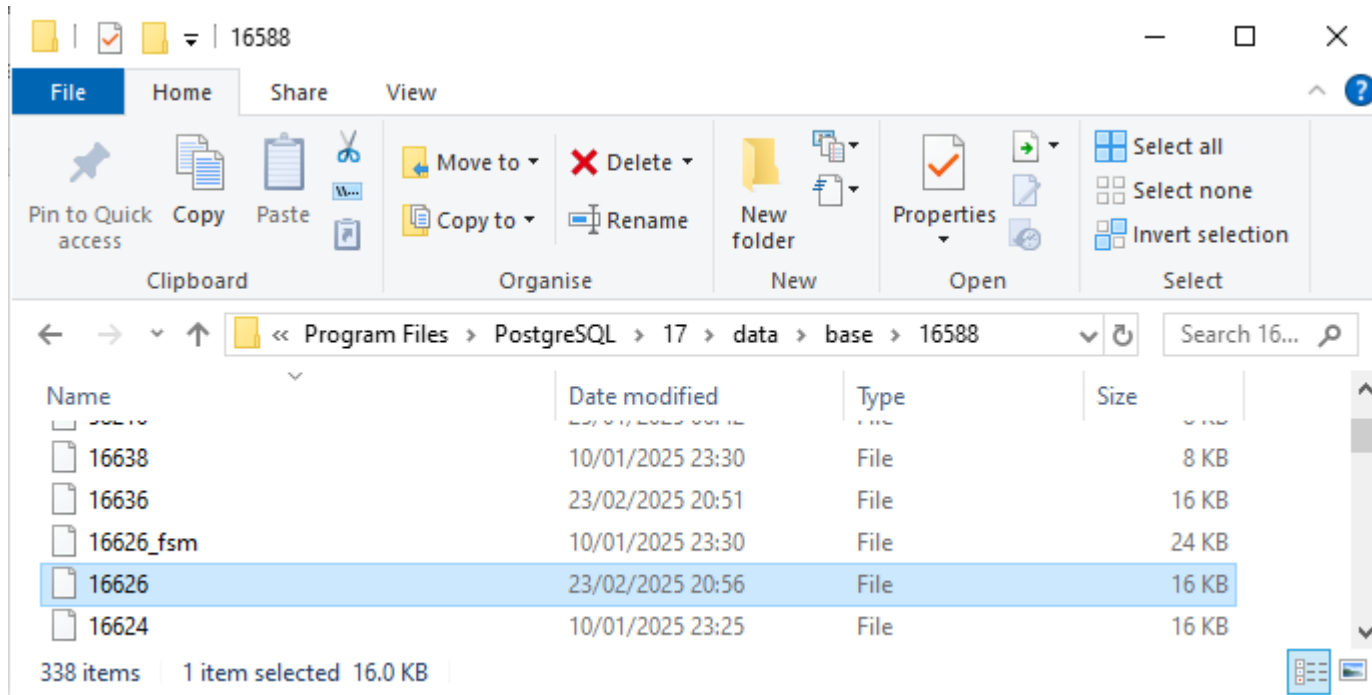


The data directory is how our database(s) are represented in a file system.

```
SELECT pg_relation_filepath('employees');
```

	pg_relation_filepath text
1	base/16588/16626

For example the employees table of the hr database is stored in folder base/16588 file 16626



It is super important not to change the contents of these files in any way or the database will become corrupted.

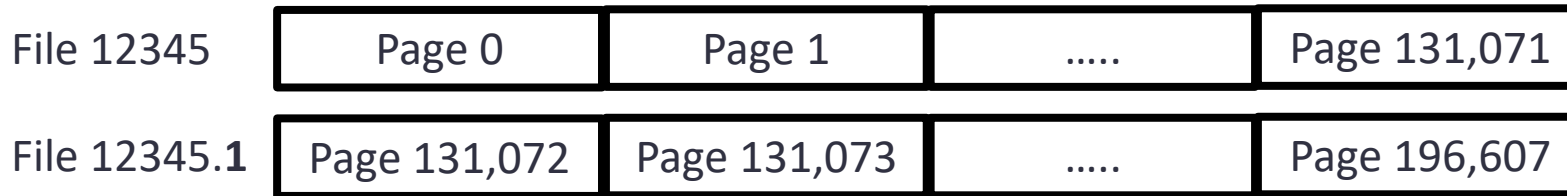
The file contains non human readable gibberish that is how PostgreSQL stores the data.

	Filename
First file	base/16588/16626
Second file	base/16588/16626.1
Third file	base/16588/16626.2

Each 1Gb chunk of the table is stored in separate files using the naming system:

Each file is further broken up into 8kb chunks called pages.

For example a 1.5GB table would be stored across two files and 196,608 pages.



Each row is stored on a single page

For large rows a technique called TOAST is used.



While processing a read query

PostGres needs to fetch a file from disk,

PostGres will read the **ENTIRE PAGE THE ROW IS STORED ON**

While processing a write query

Postgres will overwrite the entire page

43. Managing Tables

[43.1 Using SELECT INTO to create a new table with joins](#)

[43.2 Duplicate a Table with or without Data](#)

[43.3 Import Data from CSV file](#)

[43.4 Export Data to CSV file](#)

[43.5 Deleting Duplicate records](#)

[43.6 Backup Your Database Regularly](#)

[43.7 Test Queries on a Test Database](#)

[43.8 Not Documenting your Process?](#)

[43.9 Database Operations and Table Size](#)

[43.10 Tracking Table Size](#)

[43.11 PostGreSQL autovacuum Process](#)

[43.12 Recovering Unused space with Vacuum](#)

[43.13 Generate Columns](#)

43.1 Using SELECT INTO to create a new table with joins

The PostgreSQL SELECT INTO statement creates a new table and inserts data from a query into the new table.

```
SELECT
    Column_list
INTO new_tablename
FROM original_tablename
WHERE conditions
```

```
SELECT
*
INTO movies_pg
FROM movies
WHERE age_certificate = 'PG'
AND movie_lang = 'English';
```

Create a new table called movies_pg for all English movies with age rating of PG.

```
SELECT * FROM movies_pg;
```

Showing rows: 1 to 11  Page No: 1 of 1  								
	movie_id integer	movie_name character varying (100)	movie_length integer	movie_lang character varying (20)	age_certificate character varying (10)	release_date date	director_id integer	
1	11	Forrest Gump	119	English	PG	1994-07-06	36	
2	13	Gone with the Wind	123	English	PG	1939-12-15	8	
3	15	Grand Budapest Hotel	117	English	PG	2014-07-03	3	
4	22	Life of Pi	129	English	PG	2012-11-21	15	

```

SELECT * INTO top_10_movies FROM
(
SELECT
mv.movie_id,
mv.movie_name,
r.revenues_domestic,
r.revenues_international,
(r.revenues_domestic + r.revenues_international) AS "total_revenues"
FROM movies mv
INNER JOIN movies_revenues r ON mv.movie_id = r.movie_id
ORDER BY 5 DESC NULLS LAST
LIMIT 10
) as t;

```

Can we use INNER JOIN, LEFT JOIN, RIGHT JOIN, HAVING with SELECT INTO?- Yes

Create a new table called top_10_movies with most profitable movies.

```

SELECT * FROM top_10_movies;

```

	movie_id integer	movie_name character varying (100)	revenues_domestic numeric (10,2)	revenues_international numeric (10,2)	total_revenues numeric
1	48	Titanic	659.20	1528.10	2187.30
2	33	Spider-Man 3	337.00	554.00	891.00
3	31	Spider-Man	404.10	418.10	822.20
4	32	Spider-Man 2	374.10	410.40	784.50
5	35	Star Wars: A New Hope	461.20	314.20	775.40
6	11	Forrest Gump	330.30	348.10	678.40
7	22	Life of Pi	124.90	484.10	609.00
8	36	Star Wars: Empire Strikes Back	290.30	247.80	538.10
9	37	Star Wars: Return of the Jedi	309.10	166.20	475.30
10	18	Jaws	260.30	210.90	471.20

```

SELECT * INTO top_10_directors FROM
(
SELECT
mv.movie_id,
mv.movie_name,
d.director_id,
d.first_name,
d.last_name,
r.revenues_domestic,
r.revenues_international,
(r.revenues_domestic + r.revenues_international) AS "total_revenues"
FROM movies mv
INNER JOIN movies_revenues r ON mv.movie_id = r.movie_id
INNER JOIN directors d ON d.director_id = mv.director_id
ORDER BY 8 DESC NULLS LAST
LIMIT 10
) as t;

```

Create a new table
called
top_10_directors
with most profitable
movies.

```
SELECT * FROM top_10_movies;
```

	movie_id integer	movie_name character varying (100)	director_id integer	first_name character varying (150)	last_name character varying (150)	revenues_domestic numeric (10,2)	revenues_international numeric (10,2)	total_revenues numeric
1	48	Titanic	6	James	Cameron	659.20	1528.10	2187.30
2	33	Spider-Man 3	24	Sam	Raimi	337.00	554.00	891.00
3	31	Spider-Man	24	Sam	Raimi	404.10	418.10	822.20
4	32	Spider-Man 2	24	Sam	Raimi	374.10	410.40	784.50
5	35	Star Wars: A New Hope	17	George	Lucas	461.20	314.20	775.40
6	11	Forrest Gump	36	Robert	Zemeckis	330.30	348.10	678.40
7	22	Life of Pi	15	Ang	Lee	124.90	484.10	609.00
8	36	Star Wars: Empire Strikes Back	17	George	Lucas	290.30	247.80	538.10
9	37	Star Wars: Return of the Jedi	17	George	Lucas	309.10	166.20	475.30
10	18	Jaws	31	Steven	Spielberg	260.30	210.90	471.20

43.2 Duplicate a Table with or without Data

Duplicate a table with the data

```
CREATE TABLE new_table_name AS (SELECT * FROM original_table_name)
```

```
CREATE TABLE jobs2 AS (SELECT * FROM jobs);
```

```
SELECT COUNT(*) from jobs2;
```

	count bigint 
1	19

Duplicate a table without the data

```
CREATE TABLE new_table_name AS (SELECT * FROM original_table_name) WITH NO DATA
```

```
CREATE TABLE jobs3 AS (SELECT * FROM jobs) WITH NO DATA;
```

```
SELECT COUNT(*) from jobs3;
```

	count bigint 
1	0

```
DROP TABLE jobs2;  
DROP TABLE jobs3;
```

Housekeeping

43.3 Import Data from CSV file

```
CREATE TABLE countries_iso_codes (  
    country_id SERIAL PRIMARY KEY,  
    country_name VARCHAR(255),  
    iso_code_2 VARCHAR(2),  
    iso_code_3 VARCHAR(3),  
    region VARCHAR(200),  
    sub_region VARCHAR(200)  
);
```

Using a terminal window use the \copy command followed by the tablename followed by the fields in brackets then FROM 'the path to the file and file name .extension' DELIMITER ',' CSV HEADER

Header means the first row of the CSV is the header.

```
Server [localhost]:  
Database [postgres]: movies  
Port [5432]:  
Username [postgres]: postgres  
Password for user postgres:  
psql (17.2)
```

```
WARNING: Console code page (850) differs from Windows code page (1252)  
         8-bit characters might not work correctly. See psql reference  
         page "Notes for Windows users" for details.  
Type "help" for help.
```

```
movies=# \copy countries_iso_codes(country_name, iso_code_2,iso_code_3,region,sub_region)  
FROM 'C:/Users/ellio/Documents/CODING-LESSONS/15-  
Advanced_PostgreSQL/resources/countries_iso_codes.csv' DELIMITER ',' CSV HEADER;  
COPY 249  
movies=#
```

```
SELECT COUNT(*) FROM countries_iso_codes;
```

	count bigint 
1	249

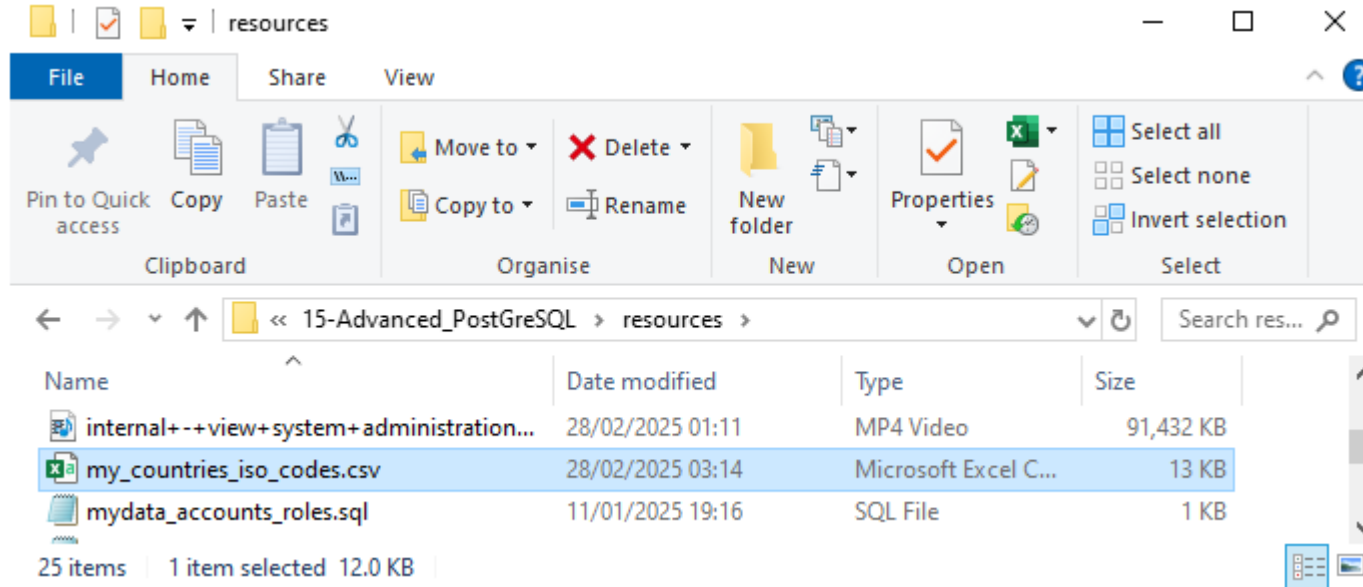
249 rows have been inserted into the table from the CSV file.

43.4 Export Data to CSV file

Use copy to export a table to CSV where the syntax is:

```
\COPY table_name TO '/location/filename' DELIMITER ',' CSV HEADER;
```

```
movies=# \copy countries_iso_codes TO 'C:/Users/ellio/Documents/CODING-LESSONS/15-Advanced_PostgreSQL/resources/my_countries_iso_codes.csv' DELIMITER ',' CSV HEADER;  
COPY 249  
movies=#
```



43.5 Deleting Duplicate records

```
CREATE TABLE colours (  
    colour_id SERIAL PRIMARY KEY,  
    colour VARCHAR(100)  
);
```

```
INSERT INTO colours (colour) VALUES  
('blue'),('green'),('green'),('red'),('yellow');
```

```
SELECT * FROM colours;
```

	colour_id [PK] integer	colour character varying (100)
1	1	blue
2	2	green
3	3	green
4	4	red
5	5	yellow

So the idea is to find the two green's.

```
SELECT  
colour,  
count(*)  
FROM colours  
GROUP BY colour  
HAVING count(*) > 1  
ORDER BY colour;
```

	colour character varying (100)	count bigint
1	green	2

One way to find duplicate rows would be to use COUNT, GROUP BY and HAVING.

```
DELETE FROM colours a USING colours b  
WHERE  
a.colour_id < b.colour_id  
AND a.colour = b.colour;  
DELETE 1  
Query returned successfully in 60 msec.
```

This is a self referencing join. I am keeping the last data with colour_id because I am using a.colour_id < b.colour_id condition.

```
SELECT * FROM colours;
```

	colour_id [PK] integer	colour character varying (100)
1	1	blue
2	3	green
3	4	red
4	5	yellow

For large tables, list all duplicates with all columns.

```
SELECT * FROM
(
  SELECT
  id,
  ROW_NUMBER() OVER (PARTITION BY column1, column2 ORDER BY id ASC) AS Row
  from table_name
) AS dups
WHERE
dups.Row > 1;
```

```
INSERT INTO colours (colour) VALUES
('green'),('green');
```

```
SELECT * FROM
(
  SELECT
  colour_id,
  colour,
  ROW_NUMBER() OVER(PARTITION BY colour ORDER BY colour_id ASC) AS Row
  from colours
) AS dups
WHERE
dups.Row > 1;
```

	colour_id [PK] integer	colour character varying (100)	row bigint
1	6	green	2
2	7	green	3

```
DELETE FROM colours WHERE colour_id = 7;
DELETE 1
```

Query returned successfully in 51 msec.

Now I can delete the duplicate

```
DROP TABLE colours;
```

 Housekeeping

43.6 Backup Your Database Regularly

BACKUP YOUR DATABASES REGULARLY

Data is hard to retrieve

especially after a power surge, a fire and other unforeseen events. Remember, sometimes the computers just stop working for no good reason, so plan it accordingly. Make frequent backups and put the backup media in a secure and safe place.

What constitute a 'safe place' depends on how critical your data is. It might be a fireproof safe in the same room as your computer. It might be in another building, or it might be on a cloud hosting platform or it might be a concrete bunker under a mountain that has been hardened to withstand natural disasters etc. You need to decide your 'Disaster Recovery Plan (DRP)' accordingly.

Always plan, so that you can execute the plan when it is needed the most!

43.7 Test Queries on a Test Database

Practice your knowledge on test or non-production server

Create a test database that has the same structure as your production database, but with only a few representative rows in the tables.

Run each test query on the test data and see whether the results match your expectations.

Build several sets of test data and be sure to include odd cases, such as;

- empty tables
- extreme values at the very limit of allowable ranges

Always try to think of unlikely scenarios and check for proper behavior when they occur

The best part is that in the course of checking for unlikely cases, you will be amazed to see several insights into problems that are more likely to happen in the real production database.

43.8 Not Documenting your Process?

Always always document everything...

The only thing you can be absolutely sure of in this world is CHANGE!

Six months from now, you won't remember why you designed things the way you did, unless you had carefully document what you did and why. Documentation means making your to capture all aspects of database designs, the logic, relationships of tables, keys, objects and much more. Put it more detail that you think is reasonable. Always document to kept this goal in mind, if you were to do this work after six months from the scratch, can you use this documentation. Always always document everything!

43.9 Database Operations and Table Size

1. Database operations on a table include but are not limited to: INSERT, UPDATE, DELETE
2. Whenever we do an 'operation' on a table, it affects table size. i.e. increases or decreases.
3. A PostgreSQL can grow as a result of routine operations. For example
 - When you update a value in a row, the database creates a NEW VERSION of that row that includes the updated value, but it does not delete the old version of the row (i.e. the dead rows)
 - When you DELETE a row, even though the row is no longer visible to you, it lives as a dead row in the table.
 - The database engine uses these dead rows to provide certain features in environments where multiple transactions are happening and old versions of the rows might be needed by transactions other than the current one.
4. To solve the above, the database engine automatically uses VACUUM to return unused space and much more.

43.10 Tracking Table Size

```
CREATE TABLE table_vaccum (  
  id INTEGER  
);
```

The table is empty and so has a 0 byte size.

```
INSERT INTO table_vaccum  
SELECT * FROM generate_series(1,20000);  
INSERT 0 20000  
Query returned successfully in 51 msec.
```

```
SELECT  
pg_total_relation_size('table_vaccum'),  
pg_size_pretty(pg_total_relation_size('table_vaccum'));
```

I continue inserting data until the table passes 1Mb in size.

```
SELECT  
pg_total_relation_size('table_vaccum'),  
pg_size_pretty(pg_total_relation_size('table_vaccum'));
```

```
UPDATE table_vaccum  
SET id = id + 1;  
UPDATE 320000  
Query returned successfully in 1 secs 33 msec.
```

```
SELECT  
pg_total_relation_size('table_vaccum'),  
pg_size_pretty(pg_total_relation_size('table_vaccum'));
```

```
SELECT  
pg_total_relation_size('table_vaccum'),  
pg_size_pretty(pg_total_relation_size('table_vaccum'));
```

	pg_total_relation_size bigint	pg_size_pretty text
1	0	0 bytes

Some test data is inserted and now the table has size.

	pg_total_relation_size bigint	pg_size_pretty text
1	761856	744 kB

	pg_total_relation_size bigint	pg_size_pretty text
1	11632640	11 MB

I run a simple update to add 1 to all the values. Note how the table size doubles because of the dead rows.

	pg_total_relation_size bigint	pg_size_pretty text
1	23232512	22 MB

43.11 PostgreSQL autovacuum Process

1. When we have a large number of rows getting deleted in a table PostgreSQL launches the VACUUM automatically.
2. AutoVacuum process is enabled by default. However you can set it to off too (not recommended)
3. AutoVacuum runs in the background without any visual interface. However you can check it's activity by a query.
4. By default the autoVacuum runs every minute.

```
SELECT
    relname,
    last_vacuum AS last_manual_vacuum,
    last_autovacuum AS last_autovacuum,
    vacuum_count AS count_manual_vacuum,
    autovacuum_count AS count_autovacuum,
    last_analyze AS last_manual_analyze,
    last_autoanalyze AS last_autoanalyze
FROM pg_stat_all_tables
WHERE relname = 'table_vaccum';
```

Note that for the table the autovacuum has run once (when I did the update all rows query), but the dead rows did not get removed and the space released back to disk.

	relname name	last_manual_vacuum timestamp with time zone	last_autovacuum timestamp with time zone	count_manual_vacuum bigint	count_autovacuum bigint	last_manual_analyze timestamp with time zone	last_autoanalyze timestamp with time zone
1	table_vaccum	[null]	2025-02-28 04:45:20.929196+01	0	4	[null]	2025-02-28 04:45:21.160877+01

```
UPDATE table_vaccum
SET id = id + 1;
UPDATE 320000

Query returned successfully in 1 secs 33 msec.
```

If I run the update query again then I would expect the table size to be 22mb + 11mb with all the new dead rows.

```
SELECT
pg_total_relation_size('table_vaccum'),
pg_size_pretty(pg_total_relation_size('table_vaccum'));
```

	pg_total_relation_size bigint	pg_size_pretty text
1	23232512	22 MB

```
SELECT
    relname,
    last_vacuum AS last_manual_vacuum,
    last_autovacuum AS last_autovacuum,
    vacuum_count AS count_manual_vacuum,
    autovacuum_count AS count_autovacuum,
    last_analyze AS last_manual_analyze,
    last_autoanalyze AS last_autoanalyze
FROM pg_stat_all_tables
WHERE relname = 'table_vaccum';
```

But this is not what is happening. Autovacuum is running to remove the previous dead rows but retains the latest dead rows.

Notice how the autovacuum count has increased from 4 to 5

	relname name	last_manual_vacuum timestamp with time zone	last_autovacuum timestamp with time zone	count_manual_vacuum bigint	count_autovacuum bigint	last_manual_analyze timestamp with time zone	last_autoanalyze timestamp with time zone
1	table_vaccum	[null]	2025-02-28 05:04:20.637857+01	0	5	[null]	2025-02-28 05:04:20.881188+01

Note that **autoVacuum** also runs the **autoanalyze** command to gather the data on the contents of the table, which is used by PostGreSQL to efficiently execute queries on the table. The more data understanding means better query execution times.

43.12 Recovering Unused space with Vacuum

```
VACUUM [FULL] [FREEZE] [VERBOSE] [table_name];  
Analyze table_name [ (col1, col2, ... coln) ];
```

FULL	Optional. If specified, the database writes the full contents of the table into a new file. This reclaims all unused space and requires an exclusive lock on each table that is vacuumed.
FREEZE	Optional. If specified, the tuples are aggressively frozen when the table is vacuumed. This is the default behaviour when FULL is specified. So it is redundant to specify both FULL and Freeze.
VERBOSE	Optional. If specified, an activity report will be printed detailing the vacuum activity for each table.
ANALYZE	Optional. If specified, the statistics used by the planner will be updated. These statistics are used to determine the most efficient plan for executing a particular query.
Table_name	Optional. If specified, only the table listed will be vacuumed. If not ALL TABLES in the database will be vacuumed.
Col1,col2	If specified these are the columns that will be analyzed.

```
VACUUM VERBOSE table_vaccum;
```

```
INFO: vacuuming "movies.public.table_vaccum"
```

```
INFO: finished vacuuming "movies.public.table_vaccum":
```

```
index scans: 0 pages: 0 removed, 2832 remain, 1 scanned (0.04% of total)
```

```
tuples: 0 removed, 320000 remain, 0 are dead but not yet removable
```

```
removable cutoff: 8174, which was 0 XIDs old when operation ended
```

```
frozen: 0 pages from table (0.00% of total) had 0 tuples frozen
```

```
index scan not needed: 0 pages from table (0.00% of total) had 0 dead item
```

```
identifiers removed
```

```
avg read rate: 0.000 MB/s, avg write rate: 0.000 MB/s
```

```
buffer usage: 12 hits, 0 misses, 0 dirtied
```

```
WAL usage: 0 records, 0 full page images, 0 bytes
```

```
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
```

```
VACUUM
```

Query returned successfully in 58 msec.

```
SELECT
```

```
    relname,
```

```
    last_vacuum AS last_manual_vacuum,
```

```
    last_autovacuum AS last_autovacuum,
```

```
    vacuum_count AS count_manual_vacuum,
```

```
    autovacuum_count AS count_autovacuum,
```

```
    last_analyze AS last_manual_analyze,
```

```
    last_autoanalyze AS last_autoanalyze
```

```
FROM pg_stat_all_tables
```

```
WHERE relname = 'table_vaccum';
```

The manual vacuum has run and found no dead rows that can be removed yet. Note how the manual vacuum count is now 1 and the time of the last manual vacuum. It also performed an autoanalyze with the manual vacuum.

	relname name	last_manual_vacuum timestamp with time zone	last_autovacuum timestamp with time zone	count_manual_vacuum bigint	count_autovacuum bigint	last_manual_analyze timestamp with time zone	last_autoanalyze timestamp with time zone
1	table_vaccum	2025-02-28 05:27:54.491952+01	2025-02-28 05:04:20.637857+01	1	5	[null]	2025-02-28 05:04:20.881188+01


```
SELECT
pg_total_relation_size('table_vaccum'),
pg_size_pretty(pg_total_relation_size('table_vaccum'));
```

	pg_total_relation_size bigint	pg_size_pretty text
1	23232512	22 MB

No space was reclaimed from a VERBOSE VACUUM.

1. When we run a vacuum without FULL, it only marks the space held by dead rows as available for future use.
2. VACUUM FULL creates a new version of the table and discards all dead rows.
3. VACUUM FULL takes more time to process, and it must have **EXCLUSIVE ACCESS** to the table which means **NO ONE CAN UPDATE** data on the table during this operation.

```
VACUUM FULL table_vaccum;
```

```
VACUUM
```

```
Query returned successfully in 262 msec.
```

VACUUM FULL has removed all dead rows.

```
SELECT
pg_total_relation_size('table_vaccum'),
pg_size_pretty(pg_total_relation_size('table_vaccum'));
```

	pg_total_relation_size bigint	pg_size_pretty text
1	11599872	11 MB

1. Each time you perform an update on a table, the original record is kept in the database. A vacuum will remove these old records (i.e. tuples/dead rows) and reduce the size of the table.
2. You can only vacuum tables in which you have VACUUM permissions.
3. You Cannot run a VACUUM command within a transaction.

43.13 Generate Columns

1. Generated columns are 'computed columns' or 'virtual columns'
2. A generated column is sort of like a view but for columns.

```
CREATE TABLE t (  
  w REAL,  
  h REAL,  
  area REAL GENERATED ALWAYS AS (w*h) STORED  
);
```

```
INSERT INTO t (w,h) VALUES (1,2),(2,2);
```

The value of area is computed at row creation time.

```
SELECT * FROM t;
```

	w real	h real	area real
1	1	2	2
2	2	2	4

This only works on a row by row basis. I.e. it cannot take values from the row above, only the current row.

When a column value is change the auto-generated column is automatically updated.

```
UPDATE t  
SET w = 10  
WHERE w = 1;
```

```
SELECT * FROM t;
```

	w real	h real	area real
1	2	2	4
2	10	2	20

1. Such functionality was earlier achieved with triggers, but with generated columns this becomes much more elegant and cleaner.
2. Persistence: Currently the value of generated columns have to be persisted, and cannot be computed on the fly at query time. The 'STORED' keyword must be present in the column definition.
3. The expression used to compute the value has to be immutable. That is, it has to be deterministic. It can depend on other columns, but no other generated columns of the table.
4. Indexes: Generated columns can be indexes, but cannot be used as a partition key for partitioned tables.
5. Copy and pg_dump: the values of generated columns are omitted in the output of pg_dump and copy table commands, as it is unnecessary. However you can explicitly include them in the COPY using:

COPY (SELECT * FROM tablename) TO STDOUT rather than **COPY TO STDOUT**

In the [lesson 26.5 Setup Presidents Speeches Data](#) we used tsvector to manually create a powerful text search column. Now we are going to automate that with a generated column.

```
ALTER TABLE docs_presidents
ADD speech_text_tsv tsvector
GENERATED ALWAYS AS (to_tsvector('english', speech_text)) STORED;
```

Course Content

the movies database overview | the actors, directors, movies & revenues tables | create movies database tables | import movies database table data | creating databases & tables in PgAdmin 4 | deleting tables from databases | inserting single & multiple rows | escaping apostrophe in a string | returning clause | delete row | delete all rows | upsert on conflict do nothing | upsert on conflict update rows | aliasing with & without AS keyword | aliasing as a string | concatenating columns in a query | order by asc & desc | order by nulls first & last | select distinct single or multiple columns or all | Comparison, Logical & Arithmetic Operators overview | And Or Operator, Combining And with OR Operator, order of execution | Where Clause Order of Operation & Alias in Where Clause | Using Comparison Operators with numeric, dates, text | Using Limit & Offset | Using Fetch | Using IN & Not IN | Using BETWEEN & Not BETWEEN | Using LIKE & ILIKE | Using Is NULL & IS NOT NULL | Concatenation, CONCAT & CONCAT_WS | Null values with ||, CONCAT & CONCAT_WS | Boolean & accepted values | CHAR, VARCHAR, TEXT differences & use cases | integers - SMALLINT, INT, BIGINT & use cases | decimal/numeric, Real & Double precision & use cases | Default behaviours of Numeric, Real & Double Precision | Date and Time - Data, Time, Timestamp & TimestampTZ & use cases | interval function w/ date/time | timestampTZ nuances | UUID Data Type | hstore & selecting from a key=>value | JSON | JSONB & GIN indexing | network address data type | Add/modify Table Columns | cast using:: | Add Check or unique constraint to a column | implicit and explicit data conversion | explicit conversion of string to int, string to bool, string to double precision | string to date, timestamp & timestampTZ | string to interval | implicit to explicit data conversions | data conversion using case statement | TO_CHAR function | TO_NUMBER function | TO_DATE function | TO_TIMESTAMP function |

Course Content

Domain Data types: address example, positive number example, email validation regex example, enum set of values, list all domain data types, Drop domain type | Create composite address object | composite inventory item example | Enum data type and dropping | alter, rename, set schema and add attribute to data type | Alter enum type in production server | Enum with default value in table | Create a Type If Not Exists Using PL/pgSQL Function | Constraints: not null | unique | default | primary key | composite primary key | foreign key | check | drop a constraint | drop a FK constraint | create sequence | advance a Sequence | get current value | set value | restart & rename sequence | use PgAdmin 4 to alter a sequence | start with | increment | MinValue & MaxValue | sequence data type | cycle sequence | delete sequence | sequence datatype | Descending sequence | cycle sequence | delete sequence | attach sequence to table column | list all sequences in a database | share one sequence between two tables | create an alphanumeric sequence | String Functions: upper | lower | intcap | left | right | reverse | split_part | trim | btrim | ltrim | rtrim | lpad | rpad | length | char_length | position | strpos | substring | repeat | replace | count() | count(*) | count(1) | count with distinct | sum | sum with distinct | min | max | greatest | least | greatest() vs max() | avg | avg with null values | combining columns with mathematical operators | to_date() | to_timestamp() | to_char() | to_date() | to_timestamp() | make_date() | make_time() | make_timestamp() | make_interval() | make_timestamptz() | extract() | date_part() | math operators with date/time/timestamp/interval | ()overlaps() | current_date(), current_time(), current_timestamp(), localtime(), localtimestamp(), now(), transaction_timestamp() | statement_timestamp() | clock_timestamp() | timeofday() | age() | date accuracy with epoch | pg_timezone_names | pg_timezone_abbrevs | show time zone | set time zone | handling date and time fields in a database | date_part() | date_trunc()

Course Content

inner join | aliasing tbl names | on clause | join clause | alias.* | inner join using | joining 2+ tables | using cast to join on dissimilar data types | left joins | right joins | full joins | self joins | cross join | natural join | using coalesce to combine data from two tables | on vs where | union | union all | union matched data types | union with filters & conditions | union with different number of columns | intersect | except | what is a schema | add,delete,alter schema | schema hierarchy | move table to new schema | schema search path | duplicate schema and data pg_dump | system catalogue pg_catalog | compare tables and columns in two schemas | schemas and privileges | int4range, numrange, tsrange, tstzrange, daterange open - closed | arrays | comparison operators with arrays & ranges | Inclusion operators with arrays & ranges | array constructions with concat ||, array_cat(), array_prepend(), array_append() | array_ndims() | array_dims() | array_length() | array_lower() | array_upper() | cardinality() | array search functions: array_position() & array_positions() | array_remove() | array_replace() | in & not in | all | any | some | array_to_string() | string_to_array | using arrays in table columns | insert & query array data | update array data | array dimensions are ignored! | unest() | multi-dimensional arrays | array vs jsonb | what is JSON? | JSON syntax | JSON & JSONB data types | representing JSON in postgresql | Creating tables with JSONB columns | select named pairs from JSONB data | update named:pairs: delete pair, update values | insert nested arrays into JSONB | remove elements from nested array in JSONB | row_to_json() | json_agg() | json_build_array() | json_build_object() | json_objects() | handling JSON nulls | jsonb_array_length() | jsonb_each() | jsonb_to_record() | jsonb_array_length() | jsonb_each() | jsonb_to_record() | existence operator | containment operator | Search JSON with postgreSQL functions | indexing JSONB with GIN

Course Content

what is an index? | create an index (single-column, multi-column) sql | pgAdmin4 | create unique indexes (single-column, multi-column) | list all indexes | index size | pg_indexes_size() | pg_stat_all_indexes() | drop an index | SQL statement execution process | SQL statement execution stages | query optimizer | optimizer node types | Query cost analysis: seq nodes | index nodes | index only nodes | Bitmap Index nodes | hash joins | merge joins | nested loop joins | working memory | b-tree index | hash index | brin index | gin index | explain statement | brin index | gin index | explain statement | explain output options | explain Analyze | query cost model & formula | cost of indexes (table size, index size, query speed) | indexes for sorted output | execution plans depend on input values | Using Organized Vs Random Data | index only vs index scan query plan speed | partial indexes | expression index | adding data while indexing | invalidating an index | re-index | what are views? | create view | rename view | delete view | create view with filters | using union joins in a view | using inner joins in a view | column order in a view | append a column to view | delete column from view? | regular view does not store data | updatable views | updatable view check option | check local | check cascade | what is a materialized view? | create matview | drop materialized view | changing materialized view data | populate matview in pg_class? | concurrent refresh mview | materialized views pros and cons | example of using matview for website stat analysis | pg query to find all matviews with no unique index | other useful pg matview queries | Subqueries | where | IN | subqueries with JOIN | using union without order by | subqueries with alias | select without from clause | correlated queries | select.. in(subquery) | any with subquery | all with subquery | subquery using exists | what is a cte | how to build a cte | using case statement in cte | join cte result set with table | simultaneous delete & insert with cte | recursive cte for parent child relationship

Course Content

simultaneous delete & insert with cte | recursive cte for parent child relationship | summarisation with grouping | group by with rollup | using grouping, coalesce & case to produce a beatified rollup table | Intro to windows functions | prepare & understand global trades sample data | use: Agg functions, rollup on single & multiple columns, group by cube & grouping sets | group by cube | aggregation query performance | filter clause | Using Window Function | over clause | sorting inside a window | ordering inside a window | sliding dynamic window | understanding window framing: rows, Between proceeding, unbound, current | rows & ranges | using multiple windows in one query | ntile buckets | lead & lag function | last_value() | first_value() | nth_value() | corr() | row_number() window with partitions | row_number() with order by | row_number() within subquery to find nth record of each partition | apply row_number() to distinct rows | window functions summary | planning tips for window functions | rank vs dense rank | compute global and group ranks | partition by group averages | order by our own defined values | delete & insert in one query using with | intro to regex | regular expressions notations table | similar to | POSIX ~, ~*, !~, !~* | Using regular expression with substring() | regexp_matches() | regexp_replace() | regexp_split_to_table() | regexp_split_to_array() | drawbacks of %LIKE% based search queries | what is full text search | to_tsvector() | to_tsquery | build tsvector table | apply GIN index on tsvector | ts_headline | ts_rank() | what is a partition? | when a partition should be used | table inheritance theory | partition types: range, list & hash | Partition by Range CRUD | partition by list CRUD | partition by hash CRUD | default partition | multi-level partitioning | add new partition | detach partition | alter partition bounds within a transaction | partition_pruning | choseing a field to partition over | dimensioning the partition | partition advantages | common partition mistakes

Course Content

postgreSQL as a development platform? | procedural languages | keep the data on the server! | functions vs stored procedures | user-defined functions | structure of a function | sql function syntax | create sql function | view function details in pgadmin4 | \$\$ and \$body\$ quotation | functions that return void | functions that return a single value | functions using parameters | a function that returns a composite (array of row data) .* or .column_name | select columns from a function return | return a table from a function | function default parameters | function based on views | drop a function | PL/pgSQL introduction | PL/pgSQL vs SQL | structure of a PL/pgSQL function | Declaring variables, using alias for | declaring variables in a function | variables initializing timing | using IN, OUT without RETURNS | scope of variables in block and subblock | how to return query results | Control Structures: IF statements, using if with table data, CASE statements (simple & searched) | loop statement | for loop | iterate over result set in for loop | continue statement | foreach loop | while loop | using return query | returning a table | return next | error and exception handling | eh – too_mny_rows | sqlstate codes for eh | data exception errors | functions vs stored procedures | create transactions | understand use of stored procedures | Returning a value from stored procedure | drop a stored procedure | intro to triggers | what is a trigger? | types of trigger | trigger table | pros & cons of triggers | trigger key points | steps to create a trigger | create a data auditing trigger | trigger to modify data at insert event | a triggered function to view trigger variables | trigger function to disallow DELETE on a table | trigger function to disallow TRUNCATE on a table | creating an audit trigger | creating conditional triggers | disallow data change on primary key | use triggers very cautiously | what is an event trigger | event triggers usage scenarios | creating event triggers | event trigger events & variables | creating and audit event trigger | prevent schema changes | dropping a trigger

Course Content

understanding row by row operations | cursors and procedural language | steps to create a cursor | creating a cursor | opening a cursor | using a cursor | updating data | closing a cursor | creating a PL/pgSQL cursor | using a parametric cursor via function | what is a crosstab report? | installing the tablefunc extension | creating a crosstab report | order matters in crosstab reports | pivoting with sample rainfalls data | pivoting rows and columns | matrix report via a query | aggregate over a filter | static to dynamic pivots | dynamic pivot query via json | dynamic pivot table columns | interactive client-side pivot | handling missing values | global character set support | client and server encoding | what is a transaction? | how SQL protects database during transaction | ACID database | transaction setup | transaction analysis | how to fix aborted transactions | how to fix transactions on a crash | partial transaction rollback with savepoints | using savepoints with transactions | how to fix aborted transaction | how to fix transaction on crash | partial transaction rollback with savepoints | using SAVEPOINT with transaction | orders shipping to USA or france | total number of orders shipped to USA or france | orders shipped to latin america | show total order amount for each order | first the oldest and latest order date | total products in each category | list products that need reordering | freight analysis | customers with no orders | top customers with total orders amount | orders with many lines of ordered items | orders with double entry items | late shipped orders by employees | countries with customers or suppliers (union & cte) | customers with multiple orders in 4 day period | first order from each country | human resources database structure | quick overview of all tables | get highest, lowest, total & average salaries of employees | difference between highest & lowest salaries by job_id | get lowest paid salaries by each manager | average salary for each department with more than 10 employees | average salary for each department excluding programmers |

Course Content

max salary for each post where salary is equal or above \$5000 | using alias name with columns | compute 15% of salaries for all employees | list all employee ids with each job id group | discard characters from employee email address | list all employees with first name beginning with A, B or M | string manipulation with upper, lower & initcap functions | using substring to get portion of string data | get unique designations in employees table | get employees with particular department id | select employees not in range | find 'c' in last name at 3rd or greater position | update phone number with strings | get monthly salary of each employee | calculate average salary with total number of employees | find employees with 6 letter names | select first or last records in a table | get the first or last record for each group | how to calculate cube root in PostgreSQL | security concepts & access levels | instance level security | add users to roles | use pgAdmin4 to create roles | database level security | schema level security | table level security | column level security | row level security | using CURRENT_USER with rls | row level policy for application users (session) | drop a RLS policy | inspecting table permissions | row level security performance | encrypted data in columns | connect to a local or remote database | switch connection to a new database | list all databases & tables | list all table space, schemas & indexes | list all sequences, roles, data types & domain data types | describe a table | edit SQL commands in an editor | display command history, run commands from a file | built in syntax reference for commands | represent NULL values on psql terminal | pretty a psql table layout | repeatedly execute a command | turn on query execution time | list database users, & database sizes with & without indexes | list all databases & schemas | list all tables and views | list all columns from a table | view system metadata via system information functions | view privileges information across tables | using system administration functions | show all running queries |

Course Content

terminate running and IDLE process | how to check live and dead rows in table | file layout of PostgreSQL Tables | Using SELECT INTO to create a new table with joins | Duplicate a Table with or without Data | Import Data from CSV file | Export Data to CSV file | Deleting Duplicate records | Backup Your Database Regularly | Test Queries on a Test Database | Not Documenting your Process? | Database Operations and Table Size | Tracking Table Size | PostgreSQL autovacuum Process | Recovering Unused space with Vacuum | Generate Columns